AFRL-VA-WP-TR-2005-3079

GEOMETRY AND GRID MODELING FOR NUMERICAL SIMULATION

John P. Steinbrenner (Pointwise, Inc.)

Todd Michal (The Boeing Company)

Pat J. Yagle (Lockheed-Martin Aeronautics Company)

J. P. Abelanet (Basis Software)

Pointwise, Inc.
213 S. Jennings Ave.
Fort Worth, TX 76104

JUNE 2005

Final Report for 23 October 2002 – 23 October 2004

STINFO FINAL REPORT

AIR VEHICLES DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

[illegible]

Nopadol Tarmallpark
Project Manager

[illegible]

Dcoulas C. Blake
Branch Chief

[illegible]

TIM J. SCHUMACHER, Chief
Aeronautical Sciences Division
Air Vehicles Directorate

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* June 2005	2. REPORT TYPE Final	3. DATES COVERED *(From - To)* 23 Oct 2002– 23 Oct 2004
4. TITLE AND SUBTITLE Geometry And Grid Modeling For Numerical Simulation		5a. CONTRACT NUMBER F33615-02-C-3249
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 0602201
6. AUTHOR(S) John P. Steinbrenner (Pointwise, Inc.) Todd Michal (The Boeing Company) Pat J. Yagle (Lockheed-Martin Aeronautics Company) J. P. Abelanet (Basis Software)		5d. PROJECT NUMBER A00H
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER 0A
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Pointwise, Inc. 213 S. Jennings Ave. Fort Worth, TX 76104		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AIR VEHICLES DIRECTORATE AIR FORCE RESEARCH LABORATORY AIR FORCE MATERIEL COMMAND WRIGHT-PATTERSON AFB, OH 45433-7542		10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/VAAC
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-VA-WP-TR-2005-3079
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited		
13. SUPPLEMENTARY NOTES This document has color content.		

14. ABSTRACT

An application programmer's interface known as API V2 was written to facilitate the sharing of unstructured meshing technology among computational analysts developing or requiring meshing methods. API V2's architecture consists of a database and a meshing library. The database library owns all mesh data and contains API functions for the transfer of data between the mesh library and the application. It also contains functions for dynamic loading of meshing libraries, allowing applications to access multiple meshing libraries simultaneously. The more compact meshing library API contains functions to extract the particulars of the library, including run-time parameters, allowable mesh types and available algorithms. Algorithm types are non-specific and may pertain to peripheral mesh operations as well as generation techniques. An example implementation of a database library is included for distribution. Its use markedly reduces the effort required to develop compliant applications or meshing libraries. An example Fortran wrapper library is also provided to allow Fortran applications to access compliant libraries. API V2's utility is demonstrated via sample implementations of two proprietary applications and a proprietary meshing library. API V2's future viability as a standard for unstructured meshing is dependent on the ability to maintain and extend the API, to provide training and to define certification criteria.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON 1Lt Nopadol Tarmallpark
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	SAR	86	19b. TELEPHONE NUMBER *(include area code)* 937-904-4050

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

Abstract

An application programmers interface known as API V2 was written to facilitate the sharing of unstructured meshing technology among computational analysts developing or requiring meshing methods. API V2's architecture consists of a database and a meshing library. The database library owns all mesh data and contains API functions for the transfer of data between the mesh library and the application. It also contains functions for dynamic loading of meshing libraries, allowing applications to access multiple meshing libraries simultaneously. The more compact meshing library API contains functions to extract the particulars of the library, including run-time parameters, allowable mesh types and available algorithms. Algorithm types are non-specific and may pertain to peripheral mesh operations as well as generation techniques. An example implementation of a database library is included for distribution. Its use markedly reduces the effort required to develop compliant applications or meshing libraries. An example Fortran wrapper library is also provided to allow Fortran applications to access compliant libraries. API V2's utility is demonstrated via sample implementations of two proprietary applications and a proprietary meshing library. API V2's future viability as a standard for unstructured meshing is dependent on the ability to maintain and extend the API, to provide training and to define certification criteria.

Table of Contents

List of Figures

List of Tables

1 Summary

Unstructured grid generation is a rapidly advancing technology among computational engineering disciplines, primarily because it permits generation times significantly faster than traditional methods. Unfortunately, new methodologies are often developed using local conventions, programming languages and formats, making the transfer of the technology among developers impractical. Recognizing this problem, the Unstructured Grid Consortium was established in 1999 as a forum for the exchange of unstructured mesh technology among various development efforts. The UGC was initially tasked with the development of an application programmers interface (API) for unstructured meshing that would facilitate the transfer of meshing technology. The first version of this API (API V1) was released by the UGC in 2002.

Continuing on that effort's success, this contracted effort was tasked with designing, developing and demonstrating a second generation API (API V2) for unstructured mesh technology transfer. This effort began with the review of related geometry, grid and computer science APIs, with the intention of identifying standards, potential collaboration with other efforts, and opportunities for reuse among these projects. With the resulting information in mind, API V1 was then carefully critiqued to identify those features of API V1 worth retaining as well as those to be discarded.

A number of recommendations for API V2 were made from these reviews. First, the API should be written in the C programming language, with function names following a well-defined convention. Second, data should be transferred to/from the application and the library piecemeal via a hierarchical function set, thereby requiring the library to be the primary owner of the data. Next, geometric data and functionality should be divorced from the API so that library developers can concentrate on meshing methods. Finally, optional data transfer should be provided so that run-time parameters unique to a given meshing method can be controlled by the application.

The resulting API V2 espouses all of these ideas within two separate libraries. The database library acts as a central data repository and is used primarily for mesh data transfer between the meshing library and the application. However, it also contains API functions for parameter (optional) data transfer, plugin functions that facilitate dynamic loading of meshing libraries, and hooks to specify geometry functions via callback registration. Plugin functionality allows users to work with multiple API-compliant meshing libraries simultaneously. An example database library of the entire database API was developed and is included with the distribution, thus significantly reducing the application and library developer's development burden. A Fortran interface library was also written (again, included in the distribution) to allow Fortran applications to link to the API reliably.

The meshing library consists of a very lean set of functions for general meshing algorithms. The library is tasked with advertising its capabilities (algorithms), mesh element requirements, and tunable parameters via API functions designed for those

purposes. Library developers are also free to include meshing algorithms beyond curve, surface and/or volume generation, such as mesh quality assessment and adaption.

The API is distributed via directly usable C header files. A comprehensive html-based hyper-linked User Manual was also developed, containing full documentation of functions and data structures, example code, explanations of terminology, possible usage scenarios, and API implementation guidelines. It also references the database, Fortran and plugin libraries distributed with the API.

The utility of the new API V2 was demonstrated via API implementations of two existing applications and a mesh library. From here, API V2's acceptance as a standard is dependent on the Air Force and the UGC's ability to encourage its usage among potential developers. This may require additional efforts such as active maintenance and extension of the API, application/library developer support and training, establishment of a repository of compliant libraries, and defining procedures for library certification.

2 Introduction

Preparation of a discretization of a three dimensional spatial region (a mesh) is a prerequisite of most computer aided engineering (CAE) software, including computational structural mechanics, computational electromagnetics, and computational fluid dynamics (CFD). Unfortunately, meshing is the current bottleneck in the CFD process, often requiring up to 75% of an analyst's labor. Until fairly recently, the inordinate amount of meshing labor was due to the prevalence of multi-block structured meshes, a relatively mature meshing methodology. Structured mesh generation involves subdividing the spatial region into contiguous, six-sided blocks, followed by discretizing each block with an ordered (i.e. structured) I x J x K array of hexahedra.

An emerging meshing methodology known as unstructured meshing has recently shown significant potential for reducing CFD meshing labor. The labor reductions are made relative to structured grids by removing the need to create a blocking topology and by eliminating the requirement to impose structure on the mesh cells. Unfortunately, the efficiencies gained from the use of unstructured meshes come with a price. The number, size, and shape of unstructured mesh cells (usually tetrahedra) are deficient with respect to hexahedra. More tetrahedra than hexahedra are required to discretize the same volume (increasing the required computing resources) and CFD solution accuracy seems to be much more sensitive to unstructured mesh cell size and shape than structured hexahedra. These deficiencies combine with the relative immaturity of the methodologies to make unstructured meshing a highly active area of research. As a result, several societies, annual conferences and symposia are dedicated to meshing [1][2].

It is somewhat of a paradox that this high rate of research and innovation in unstructured meshing tends to stifle the deployment of new methodologies in meshing applications. Though new methodologies are being developed by a variety of organizations from industry, government, and academia, each implementation is usually developed following local programming practices. Once published, significant software re-engineering may be required to implement a new meshing library into an existing application. Although motivated by the desire for a state-of-the-art meshing application, an application developer may find that the re-engineering effort negates any potential gains from use of the new method. Consequently, new unstructured grid technology is slow to be incorporated into existing applications, and the sharing of new technology among mesh library developers is minimal.

One approach to overcoming these difficulties is to develop a specification that governs the interaction between meshing software components. Software that follows this specification would be guaranteed to work with other software conforming to the specification. A prerequisite for this approach to be successful is that the specification be accepted and implemented by a sufficient base of software developers. There are many software specifications that have enjoyed this kind of success. Notable examples include the Object Management Group (OMG) [3] and the CFD General Notation System (CGNS) [4] efforts. These specifications have been successful in part because they were designed by a team of software developers representing a cross-section of the potential audience.

The Unstructured Grid Consortium (UGC) was formed in 1999 as a forum for facilitating the interchange of unstructured grid technology between various development efforts [5]. This alliance has grown to include members from industry, government labs, universities, and small companies. This group is tasked with coordinating the development and implementation of an Application Program Interface (API) to govern the interaction between mesh generation software modules or libraries and mesh generation applications. Version 1.0 of this specification (API V1) was completed in 2002 [6]. Shortly after this API was released, the need for further development of and extensions to the API was well understood. Unfortunately, many of the participating agencies in the UGC were unable to secure additional internal funding for this project, which limited the opportunity for further extensions. Recognizing the risk of allowing the project to stall just as it was taking shape, the U.S. Air Force released a PRDA in mid-2002 for contracted continuation of the API development. The results of the ensuing two-year effort, conducted by Pointwise, Inc., with Boeing and Lockheed-Martin as subcontractors and Basis Software as a consultant, are presented in this report.

The overall objective of this effort is the design, development and demonstration of a new API (API V2) for unstructured meshing software that will further reduce the burden associated with meshing software integration and sharing. Successful adoption of API V2 as an industry standard will provide many benefits. It will accelerate the transfer of mesh generation technology from the researcher to the user and reduce software development and support costs. Government and university researchers may use this API to share mesh generation modules, thereby accelerating development of new technology. Finally, an API standard may open new markets for commercial software vendors looking to develop API-compliant frameworks and modules that can be marketed together or individually.

The document is roughly divided into two parts, the first corresponding to the review and analysis of previous and related efforts, and the second corresponding to the new API designed and developed as a result of this analysis.

Chapters 3 through 6 describe the review of the original API V1 and related efforts that were conducted during the first half of the effort. These reviews were used to forge recommendations for API V2 (Chapter 5) and also identified opportunities for coordinate efforts with other projects (Chapter 6).

Chapters 7 through 9 describe the new API that was developed primarily in the second half of this effort. Chapter 7 outlines the design of API V2, including conventions, scope, terminology, architecture and usage scenarios. Chapter 8 presents several demonstrations of the API, describing example implementations of key features as well as full implementations of the library using commercial and proprietary grid generation software. Finally, Chapter 9 discusses the API V2 deliverable items of this contract.

A concluding chapter summarizes the effort and offers recommendations for future versions.

3 Review of Related Efforts

In order to design an unstructured grid generation API that is applicable to a wide range of engineering analyses, a broad awareness and understanding of the tools available to analysts was first required. Toward this end, an extensive literature and web search for existing and emerging grid generation standards was conducted. The search was initiated by starting from known grid generation repositories, expanding outward following references and hyperlinks.

The results of these searches helped to resolve a number of issues that influenced the design of API V2. The more important of these issues are described below.

- **Justification –** Even a cursory literature search should answer the basic question of whether or not an accepted API for unstructured grid generation already exists, thereby justifying (or not) the need for a new one.

- **Alternate Design Formats –** Careful examination of related APIs provides exposure to formats, languages, syntaxes and practices that might not otherwise be considered. This creates the potential for revolutionary, rather than evolutionary changes to be incorporated in the development of a second generation API. Such changes would likely be employed to facilitate ease of use, efficiency, and familiarity.

- **Accepted Industry Standards –** Patterning the API after accepted standards in similar and related fields increases the likelihood of the API’s short-term acceptance and long-term viability. It also potentially eases the burden of incorporating the API into existing software, and promotes compatibility with other efforts.

- **Existing Components for Reuse –** It is possible that this API may have direct applicability with other ongoing or completed efforts, either as a subset or superset. For example, this API may be compatible for use as a subset of a general analysis API, or a low-level data manipulation API may be usable as a component of this effort. In either case, API reuse could significantly reduce the overall effort of one of the two parties. Also, existing libraries may prove to be candidates for demonstration of the new API.

- **Collaboration With Other Efforts –** Even if a natural fit between efforts is not found, it may still be possible to identify ongoing efforts with which mutual collaboration would be advantageous to each.

With these five elements in mind, a total of sixteen related efforts were identified and reviewed, divided into four different categories. These reviews were completed in June, 2003, and hence may already be dated, incomplete, and/or inaccurate. For this reason, references to current URLs for each project are provided.

Finally, these reviews are not intended to be subjective in nature. No recommendations or rejections may be implied from them; they are intended only to help in the design of API V2.

3.1 Meshing APIs

For obvious reasons, APIs for meshing were the most important class to identify and review. A total of five meshing APIs were identified.

3.1.1 Algorithm Oriented Mesh Database

The Algorithm Oriented Mesh Database (AOMD) [1], developed at Rensselaer Polytechnic Institute (RPI), is a method of describing the connectivity of a mesh. It is written entirely in C++ and is available freely under a license similar to the GNU GPL. AOMD provides an efficient method of maintaining and accessing connectivity information for a mesh, while at the same time reducing the memory requirements for the storage of a mesh and its connectivity. It also allows for mesh adaption and various element types.

AOMD has the potential to provide an efficient framework for describing mesh connectivity and reducing the memory required for mesh storage. It also uses the established Standard Templates Library and may be reused as long as the source is provided without charge and a disclaimer is included. However, existing documentation is limited, and an on-line manual is not yet available.

3.1.2 Field Model

Field Model [8] was developed by Patrick Moran at NASA ARC as a second pass at the Field Encapsulation Library (FEL). The source is written entirely in C++ and is freely available under the MIT license. Field model uses an abstraction of the general mesh problem to represent mesh and field data on both structured and unstructured grids. For example, the **`faces`** method within the **`FM_mesh`** class is used to return all entities of higher and lower topological value. Applied to a surface, **`faces(1)`** will return a list of all edges using it, **`faces(2)`** will return the surface itself, and **`faces(3)`** will return all volumes using it.

The techniques used in Field Model are probably best suited for the transfer and interpretation of field data rather than the mesh itself. Initial software timings of Field Model indicate speeds that vary from equivalent to direct data structure access to an order of magnitude slower. Little attention appears to have been given to efficiency at this time. A web search revealed little usage of the library.

It may be advantageous to borrow some general element storage and manipulations concepts for the GGMNS API. This would provide for consistent and general argument lists in the API, but it likely would require significant effort on the library developer's part.

3.1.3 Grid Algorithms Library

The Grid Algorithms Library (GrAL) [9] is a method of coding mesh operations generically using templates. It was developed by Guntram Berti, now with NED C&C Research Laboratories, and appears to be a continuation of his Ph.D. dissertation. GrAL is written entirely in C++ and is freely available, though a cursory web search revealed minimal references to the library.

GrAL uses modern programming techniques, such as the Standard Templates Library and generic programming. It is an example of generic programming in that the abstract functionality of the application is separated from the specific details of a given case.

The GrAL kernel contains function of 4 classes: Combinatorial, Geometry, Grid and Partial. The Combinatorial functions allow access to subset and superset components of the grid (incidence iterators), as well as adjacent-component information (adjacency iterators). Geometry functions include information about vertices and volumes. Grid functions act on the entire mesh. Lastly, the partial functions include storage for subsets of entities.

Several general-purpose tools have already been implemented into GrAL. A Cell Neighbor Search algorithm is designed to find a cell’s neighboring cells. Subrange and Closure Iterators are used for marking strategies to visit each element of a given type in a given range. Boundary Iterators use switch operators, incidence information and marking procedures to traverse the boundaries of manifold-with-boundary grids. Vertex merging, currently implemented via direct node-to-node comparisons, is used to allow nodes of disjoint grids to be geometrically united. The current algorithm could easily be optimized.

An example is provided which demonstrates how to merge structured and unstructured 2d meshes into a hybrid mesh. Starting with an inner structured mesh generated using Knupp's ortho mesher, the CopyGrid function is used to load the mesh into a hybrid structure. The outer boundary of the hybrid mesh is then extracted via boundary iterators and used as input to Shewchuk's Triangle mesher. The two meshes are then joined into a hybrid mesh using the **`EnlargeGrid()`** function, and the internal vertex-merging algorithm is employed to identify common vertices.

Problems commonly associated with generic programming are identified within the documentation. For instance, generic programming techniques are difficult to use, compile and debug, which makes them particularly vexing to beginners. In addition, they are expensive in terms of memory requirements and compile times.

3.1.4 TSTT

The goal of the Terascale Simulation Tools and Technologies (TSTT) [10] effort is to create interoperable and interchangeable meshing and discretization software. The effort seeks to formulate a broad, comprehensive design that encompasses many varied aspects of the meshing and discretization process, including advanced meshing technologies, high-order discretization techniques, and terascale computing issues.

The TSTT Inter-Operable Meshing effort is focused on developing a common interface for mesh, geometry and topology access. This interface does not require an application to use a particular data structure or implementation. Accessor interfaces are used to view data. The challenge of the effort has been to balance performance with the flexibility to support a wide variety of mesh types and the desire to keep the interface simple. The TSTT group is working closely with the Common Computer Architecture (CCA) forum to investigate the use of SIDL/Babel language interoperability tools. SIDL is being used to define interfaces in a language-independent manner. Bindings for C, C++ and Fortran have been created using SIDL for mesh implementations. Ongoing tests are looking at the performance impact of using SIDL, especially in fine-grained operations such as entity-by-entity access. In addition, efforts have been initiated to merge two TSTT tools, Overture and Frontier, and a version of the MESQUITE mesh quality component is under development that will adhere to the TSTT interface definition.

The TSTT mesh interface uses a hierarchical representation of the mesh, with Level A containing geometry information, Level B containing the full mesh, and Level C containing mesh components. Access to the mesh is through these different levels. At Level B, code developers can access the entire grid hierarchy as a single object, and call functions that provide (e.g.) partial differential operator discretizations, adaptive mesh refinement or multilevel data transfer over the entire mesh. In addition to the high level interface, access is provided to the hierarchy at low levels to facilitate incorporation of tools into existing applications. This could provide access to Fortran-callable routines that return discretization or interpolation coefficients at a single mesh point. The interface standard requires that all implementations must provide high- and low-level access to the mesh.

The mesh itself is divided into entities named **`VERTEX`**, **`EDGE`**, **`FACE`** and **`REGION`**. Supported face types include triangle, quadrilateral and general polygon. Tetrahedron, hexahedron, prism, septahedron and general polyhedron volume elements are supported in 3D. Coordinate and adjacency (connectivity) information is provided on an entity-by-entity basis for the entire mesh as a collection of arrays. This data can be obtained in chunks of user-specified size (e.g. 100 entities at a time). The interface definition currently supports mesh creation, loading, destruction and services to access basic information such as geometric dimension, number of each entity, etc. Users may also attach data, such as boundary conditions, to any mesh entity.

A discussion with Lori Freitag of the TSTT effort provided further information. The team has been working primarily on interface definition. Interfaces consist of lists of parameters to be passed and definitions of query functions to access the data. Data is stored in the component, while opaque pointers to the data are used to pass through interfaces. The current focus is on low-level interfaces (e.g., add a vertex or an element), though future work may focus on high-level interfaces such as mesh generation. All of the TSTT interfaces follow SIDL/Babel rules.

The implementation of a new module/component involves developing a wrapper that matches the interface definition, followed by providing query functions for accessing and/or returning data. TSTT contributors at Rensselaer Polytechnic Institute (RPI) are developing a standard reference set of query functions, which will satisfy the majority of new modules' querying needs.

The discussion also covered the current status of the TSTT effort. The mesh query interface is well defined and exists for three different objects, namely mesh, geometry and field data. Mesquite, a grid quality improvement code, has been retrofitted to use the interface definitions. Simple example cases are under development at RPI and should be available when complete. The TSTT group is interested in the possibility of using the GGMNS interface as a high-level interface definition for TSTT.

Working collaboratively with the TSTT group could be advantageous. It would allow reuse of some of the existing TSTT body of work within the GGMNS effort. A common high-level grid generation interface could be developed either by making the GGMNS API compliant with TSTT standards or by writing a wrapper around the GGMNS API. The SIDL/Babel language interoperability tools could be re-used in the GGMNS effort. There is also the potential for commonality between the TSTT accessor functions and the GGMNS data retrieval interfaces. Compatibility between the GGMNS and TSTT efforts would benefit both in terms of the number of available tools.

3.1.5 GNU Triangulated Surface Library

The GNU Triangulated Surface library (GTS) [11] is a collection of software utilities for the manipulation of triangulated surface representations. It was developed entirely in C and is available under the GNU software license.

Several functions have been incorporated into the GTS library. They include 2D Delaunay triangulations, set operations on surfaces, surface refinement and coarsening, dynamic continuous level-of-detail, utilities for efficient point location and collision/intersection detection, graph partitioning, metric operations (area, volume, curvature), and triangle strip generation for fast rendering.

All procedures within GTS use an implied object-oriented data structure. Classes are defined for Points, Vertices, Segments, Edges, Triangles and Faces. Class hierarchy is implemented via nested C structures and type casting. Interface definitions are built around a specific object-oriented model.

GTS requires that all library functions use the GTS object-oriented model. It also requires that the GTS library data structures and object-oriented model be implemented in the calling applications. If this is not possible or desirable, file I/O may be used to transfer data to modules. This requirement implies that modules must be developed with GTS in mind, making retroactive implementation difficult. These requirements may make GTS too restrictive to use as a general standard.

For the GGMNS project, however, it may be possible to examine the GTS data structure for inclusion in the GGMNS interfaces. It may also be possible to write wrappers around some GTS functions as part of an API-compliant meshing library, which could then be documented as an example application. GTS may also be a good starting point for collecting public-domain modules that are compatible with the GGMNS standard.

3.2 Geometry APIs

Since the CAD industry is significantly more mature than the analysis field, it is logical to search for API standards already existing in that field. with the intent of adopting some of these standards in a grid generation API. Geometry APIs are also of great interest due to grid generator's reliance on CAD models for surface meshing. A total of five geometry APIs were identified and reviewed.

3.2.1 Acis

ACIS [12] is an object-oriented three-dimensional modeling engine commercially offered by Spatial Technology. It is a B-Rep solid modeler allowing manifold and non-manifold geometries/topologies. All linear and quadric geometries are represented analytically, while free-form geometries are represented using NURBs. All surface types, including B-Spline, implicit, and foreign surfaces, are bi-parametric and can be evaluated up to the second derivative.

The ACIS API is available in either C++ or Scheme. Function wrappers for C and Fortran are not provided, but C wrappers should not be difficult to develop. The API is full-featured, and includes modules that perform rendering, covering (skinning), blending, laws, Boolean operations, tolerant modeling, cellular topology, and many others.

Basic usage of the ACIS API requires thirty to fifty dynamically linked libraries (DLLs) and more than thirty megabytes of free RAM. CAD translator modules, for many major CAD packages, are available as a separate product for use with ACIS. Also available separately is a Deformable Modeling Module.

ACIS could serve as a foundation for a geometry library, but the fact that it is written in C++ would require C wrappers to be written for all API calls needed in the library. This potential could be dampened by the fact that ACIS is a commercial product. The interface structure should probably be examined in more detail to provide ideas pertaining to the GGMNS API structure.

3.2.2 Parasolid

Parasolid [13] is a commercial solid-modeling kernel. It serves as the basis for Unigraphics and several other CAD/CAM/CAE packages. It is an exact boundary representation (B-Rep) solid modeler. All surface types, including B-spline, implicit and foreign surfaces are bi-parametric and can be evaluated up to the second derivative. C-language bindings are provided, although C++ and Fortran bindings should be possible as well.

As the geometry foundation for a modular meshing application that is independent of the modeling kernel used, Parasolid could serve as the basis for surface and volume meshing algorithms that work in surface parameter space (2-D), as well as for those that work in model space (3-D). Functions within Parasolid that evaluate surfaces and their derivatives could be used as templates for the geometry callback typedefs in the GGMNS API. As an example of a successful library with an object-oriented interface, the Parasolid interface structure could be examined for potential ideas on structuring the GGMNS API.

3.2.3 Open CASCADE

Open CASCADE [14] is a collection of open-source modules and a framework for the creation and manipulation of geometry objects. It is maintained by Open CASCADE S.A, a company also offering custom development and technical support. Existing modules include foundation classes, primitive types, memory management, exception handling, classes for manipulating aggregates of data, math tools, ASCII file input and output, and the Cascade data model, which are data structures for representing two- and three-dimensional geometry and topology.

OPEN CASCADE contains a number of capabilities typical of general CAD systems. These include geometric tools for creating and modifying two- and three-dimensional geometry, intersection and projection of entities, sewing of entities into solid representations, mesh algorithms for shape triangulation, visualization tools, shape healing, and data transfer functions such as STEP/IGES readers and direct connectors to CAD packages.

The Open Cascade Application Framework (OCAF) is an open-source framework to build around existing applications. It includes geometry, input/output, creation, manipulation, and visualization tools. OCAF provides a C++ framework and infrastructure for developing new geometry-based applications. The GUI framework/infrastructure is intended for developing a GUI for applications based on OCAF. Applications using OCAF must use the Cascade data model.

Open CASCADE could potentially be used as an example implementation of geometry callbacks in the UGC API. Since it is open source, this demonstration might be releasable to the general public. The Open Cascasde Application Framework could also be used to build a demonstration application around the UGC API. Since OCAF is open source, that resulting code could be released as well for use as a tutorial to future application developers.

3.2.4 CAPRI

The Computational Analysis Programming Interface (CAPRI) [15] is a geometry-based infrastructure for analysis and design. It consists of a CAD vendor-neutral API, the CAPRI geometry kernel, and the CAPRI geometry database. All geometry evaluations and calculations are performed by the CAD system itself, with CAPRI acting as an interface between the application and the CAD system. The geometry is required to be in the form of a solid model and the CAD system must be running when using CAPRI.

The CAD developer APIs currently supported are Parasolid (UniGraphics and SolidWorks), ProToolKit (ProENGINEER), CATGEO (Catia 4.2) and CV-DORS (CADDS).

The CAPRI "standard" geometry and topology model consists of nodes, edges, loops, faces, boundaries and volumes. An edge is a parameterized curve. A loop is an ordered collection of edges. A face is a parameterized surface bounded by 1 or more loops. A boundary is a collection of 1 or more faces. Finally, a volume is a closed collection of boundaries.

The CAPRI API consists of the following components: utility routines for loading and saving parts and assemblies, database queries for obtaining the details of any geometric entity, point queries to allow placing points on a geometry, geometrically derived queries such as the length of an edge, boundary data routines for control over discretizations, attachment and interpolation routines; tag routines; and geometry creation and modification routines.

CAPRI provides separate APIs for Fortran and C/C++. The C interfaces return pointers to memory blocks. The Fortran calls fill arrays from the calling routine, where sizes must be passed in the call. The calls return simple arrays and constants of standard types. No structures are used.

C example:
```
Icode=gi_dTesselFace(vol, face, *ntri, **tris, **tris,
*npt, **pt, **ptype, **pindex, **uv )
```

FORTRAN example:
```
ICODE= IG_DTESSELFACE(VOL, FACE, NTRI, TRIS, TRIC, NPT, PT,
PTYPE, PINDEX, UV )
```

Tags and Attachments are used to pass data between modules. Tags place simple quantities associated with the volume or boundary of interest into the CAPRI database. Attachments are used to map complex quantities to boundaries. The application is responsible for filling the required tags for the target module.

Examination of the CAPRI API may provide guidance in creating a list of functions that should be included in the UGC geometry callbacks. CAPRI could be used as a demonstration case to satisfy the geometry callbacks in the UGC API. Since CAPRI is widely used, this demonstration could form the foundation of a CAPRI callback library to be used by future developers wishing to utilize CAPRI with UGC compatible grid generation libraries.

3.2.5 CAD Services - OMG

The Object Management Group (OMG) [16] is a consortium that produces and maintains computer industry specifications for interoperable enterprise applications. The CAD Services specification of the OMG was developed to define high-level engineering interfaces for answering queries of CAD data. The interfaces are based on CORBA to avoid the problems associated with data translation. All database queries use native CAD system geometry kernel, and client-server operation is possible.

The CAD Services-OMG has the following features: geometry and topology queries for manifold and non-manifold geometries, parametric regeneration of solid models, tagging of geometric entities with application specific information, basic geometry creation, tesselated representations of geometry, and access to assembly structure of CAD models.

Use of the CORBA ORB in the CAD Services-OMG specification can add significant computational overhead. It may be possible to minimize the overhead by designing interfaces to provide data in substantial quantities (coarse grain rather than fine grain).

The OMG CAD services spec could be used as a sample application of the API geometry callbacks. This demonstration could form the foundation of an OMG callback library compatible with the API. This library may be useful to future application developers wishing to use the OMG CAD services spec in conjunction with UGC compatible grid generation libraries.

3.3 Geometry/Meshing APIs

Of all the APIs reviewed, only one encompassed both geometry and mesh generation.

3.3.1 GGTK

The Geometry and Grid Toolkit (GGTK) [17] is a library of geometry modeling and grid generation functions developed and maintained by the Enabling Technology Laboratory of the Mechanical Engineering Department at the University of Alabama at Birmingham. It consists of three main categories of modules. The Geometry module contains functions for geometry creation and modification defined by NURBs curves, surfaces and volumes. It also contains many geometric utility functions. The topology module consists of data structures and functions for topological representation of geometry in terms of vertices, edges, faces and volumes. The topological representation is used for "watertight" geometry, which is a requirement for meshing. Finally, the grid module deals with the generation and manipulation of various types of grids – structured and unstructured meshes over, 2-D, surfaces and volumes. It also contains functions for curve packing, transfinite interpolation, elliptic grid generation, grid transformations, structured grid quality and grid interpolation.

The libraries are written in C++, and thorough documentation (formatted by Doxygen) of all modules exists on the project website. This documentation includes the data structures for supported element types. Availability of the software itself is unknown.

In this library, grid components are created directly on topological elements (edges, faces and volumes), which explain the requirement for watertight representations. Geometries can be imported from IGES files or can be constructed internally. Resulting edge, surface and volume meshes are then exported to file in an unpublished format. Geometry entities can also be exported in XML files.

This library represents a powerful toolkit for very specific grid generation operations. Though the requirement of a watertight geometry is very restrictive, it does allow for nearly automatic unstructured meshing in many cases. However, faulty or incomplete geometry representations will need to be repaired prior to mesh construction. Also, since the meshes are formed directly on the geometry (thereby inheriting its topology), there is no apparent provision for controlling grid point distribution on the interiors of grid elements (edges, surfaces, volumes). Though this effort is developing an API for meshing (as well as an implementation thereof), its use is probably too narrow in scope for collaboration with the GGMNS API.

3.4 Computer Science APIs

A total of five APIs pertaining to general-purpose computer science issues were identified and reviewed. These APIs were reviewed primarily to identify modern computer science practices, with the intent of applying some of these techniques into API V2.

3.4.1 Standard Template Library

The Standard Template Library (STL) [18] is a generic C++ library that provides both data structures and basic algorithms for core storage and manipulation of data. It uses the concept of a container class to store groups of data including, but not limited to, vectors and lists. Algorithms are available for sorting and interrogation of container objects. The STL also provides iterator functions. The container classes and associated algorithms appear to support multi-threading for data access, but not for data modification. STL supports all major Unix, Linux, and Win32 platforms.

STL is a well-established generic library, and any use of it within this effort would be at the generic data manipulation level. It would be difficult for the GGMNS effort to influence STL's development at all. While direct use of the powerful container and iterator concepts would be difficult due to the required conversion to an ANSI C form, their capabilities could be examined in more detail in order to identify necessary functionality in the GGMNS API.

3.4.2 Component Object Model Technology

Component Object Model (COM) [19] is a binary standard for software object interaction. It is developed and maintained by Microsoft, and serves as a core software technology in Microsoft Message Queuing (MSMQ), ActiveX controls, and others. It is primarily targeted toward Windows-based platforms, but is also available for the Unix platforms (not as freeware) including Solaris, SGI and DEC/Compaq. The implementation of COM is designed to use C++ functionality extensively. It appears possible to write C wrappers and interfaces, but they would be neither clean nor trivial.

Using COM, each object or component is treated as a "black box" with strong encapsulation. Modification and revision of modules is only allowed through the creation of new version of the module. COM allows calls to components on different computers.

Since COM is a core Microsoft technology, it is effectively an industry standard for Windows-based operating systems. This also makes it less useful for the GGMNS API since the ports to Unix operating systems are less likely to be supported fully. In addition, COM is written in C++ while the GGMNS API is implemented in C. However, COM's use of strong encapsulation and rigorous API standards are aspects that should be considered when defining the GGMNS API.

3.4.3 Common Component Architecture Forum

The Common Component Architecture (CCA) [20] is developed by the CCA-Forum, a large, organized group that provides forums, meetings and voting structure for the API. CCA is a large effort, having received $16M from the Department of Energy (DOE) over five years. The goal of CCA is the development of a software architecture based on software components and a set of computer-science rules which govern the interaction between components. It is similar to CORBA or COM/DCOM but is oriented towards high-performance computing. The major differences are the minimization of performance overhead, support for parallel and distributed execution models, and the support of High Performance Computing languages (C, C++, Fortran and Fortran 90). The goals of CCA are interchangeability, promotion of reuse, and plug-and-play.

Within the framework of the CCA, a component is defined as a binary unit of independent deployment, separated from other components with no dependence on global data. A component interacts with the environment through interfaces call ports. A component may either provide a port, which means that it implements the class or subroutines of a port, or may use a port, which means that it calls methods or subroutines in a port. A component comes with clear specifications of what ports it requires and provides. In object-oriented languages, a port is a class or interface. In Fortran, a port is a group of subroutines or a module.

Component architecture is defined as a set of standards and a framework that holds and runs the components and provides services, which allow the components to interact with other components. While the framework is embedded in the application, the standard CCA services library provides most of the tools necessary to integrate the framework into the application. These services include constructor and destructor methods, as well as a method to tell the framework which ports it uses and provides. Each component must explicitly publish with the framework which capabilities it provides and what connections it requires. All component ports are defined in the Scientific Interface Definition Language (SIDL) to provide language independence. The Babel software then transforms the SIDL definitions into language-specific stubs.

The focus of CCA is on large-scale, or coarse, integration of components for scientific computing. The concept of components provides modules, which are easily interlinked to solve large scientific computing problems. The framework allows for calls to components on different computers and is responsible for component-to-component data transfer. It does not, however, provide a mechanism for the transfer of large data sets. The framework does incur an overhead penalty due to the additional level of communication. Each call takes approximately 2.7 times as long as a direct C or Fortran call. This overhead might be too costly for fine-scale interfaces.

The CCA is a large group whose goal seems to be very similar to that of GGMNS. Their approach and API definition are both compatible with the GGMNS API. The CCA approach could serve as an example for GGMNS. Collaboration between CCA and GGMNS could be useful, but it is unlikely that the GGMNS effort would be able to influence any changes at this stage of the CCA development. Instead of collaboration, the GGMNS API could adopt a portion of the CCA standard and keep current with that standard.

In developing the GGMNS API, the use of SIDL for interface definitions is worth investigating. At first glance the approach seems very complex. Most of the complexity comes from the use of the Babel software. The advantage of the Babel software is that it allows the interfaces to utilize Object Oriented data structures and to work between multiple programming languages. The CCA approach entails additional overhead beyond the UGC version 1.0 specification primarily because of the software libraries that must be embedded into the application, to handle the SIDL/Babel interfacing and to support the CCA framework services. This complexity may cause some to avoid the adoption of the GGMNS API. A closer look at Babel may reveal that it could be used just to provide the language-specific stubs, which could then be used independent of the Babel runtime libraries.

The concept of "using" and "providing" ports would have advantages when dealing with multiple components that perform the same function within a single application. The downside is that the application or framework must keep track of the components and registered ports. The CCA framework services may be able to take care of this. Inclusion of standard software libraries in the GGMNS API would complicate delivery and maintenance of the API. Further, questions would arise as to where the responsibility lies for delivery and maintenance of the standard libraries (CCA).

3.4.4 Babel/SIDL

SIDL [21] uses the Interface Definition Language (IDL) to provide a language-independent interface to an application. SIDL (Scientific IDL) is an extension to the IDL, which adds support for complex numbers, dynamic multi- dimensional arrays, parallel communication directives, enumerated types, symbol versioning, name-space management, and object-oriented inheritance. Babel is software designed to interface software components and provide run-time libraries.

In the SIDL/Babel framework, each API component interface is defined using SIDL, and then Babel uses the SIDL description to generate glue code for each of the supported programming languages. Babel consists of a SIDL parser generating intermediate XML representations of interfaces, a code generator which reads the intermediate XML and generates the glue code, and a small run-time support library. The SIDL/Babel framework is responsible for component-to-component data transfer.

SIDL/Babel is maintained by a large, organized group providing forums, meetings and voting structure for the API. Its focus is on large-scale (coarse) integration of components for scientific computing. It employs the concept of components to provide modules, which are easily interlinked to solve large scientific computing problems. The concept of components is very similar to the COM and STL standards.

SIDL/Babel supports calling libraries written in C, C++, F77, Python, or Fortran90 (beta). It supports drivers written in C, C++, F77, Python, Java or Fortran90 (beta). It is supported currently on Linux and Solaris platforms, but allows calls to components on different computers. SIDL/Babel is freely available and licensed under the Lesser GNU Public License. SIDL/Babel is being used by the Common Component Architecture (CCA) effort.

There would be several advantages to employing SIDL/Babel in the GGMNS API. The IDL is based on a well-established standard and is part of the OMG CORBA effort. Its use may also help make interoperability with TSTT components possible. It would also offer the ability to specify the API generically. Finally, it could be used to provide inter-language operability for the GGMNS API.

Several disadvantages exist as well. For fine-scale operations, the overhead of the Babel framework and required data conversions would be very costly. It is also dependent on several supporting libraries (Java, libtool, etc.) during the build process. This introduces an additional step in the interface development process.

3.4.5 PETSc

PETSc [22] is a suite of data structures and routines for parallel implementation of applications solving partial differential equations. The suite includes parallel linear and nonlinear equation solvers as well as time integrators. PETSc is divided into libraries for manipulation of a family of objects. Each library consists of a set of calling sequences using particular data structures. PETSc may be used in Fortran, C and C++. Interfaces are written in C and Fortran versions are handled as wrappers around the C interfaces. The

Fortran wrappers are similar to the approach used in Version 1 of the UGC API, which consisted of preprocessor directives. PETSc interface procedures could be examined for inspiration in designing the GGMNS API.

3.5 Recommendations and Conclusions

The results of these sixteen reviews provide sufficient information to address the basic issues raised at the beginning of this chapter. They also serve as design guidelines for the new API.

3.5.1 Justification

The principal objective of this effort is to develop an API for general unstructured grid generation and related tasks that will allow different grid technologies (libraries) to be transferred easily among grid generation applications. A secondary design goal is to keep the API lean and easy to use, in order to maximize its likelihood of adoption by others.

None of the six meshing API/library efforts identified in this task appear to meet both of these objectives. AOMD is library of functions for describing the connectivity of a mesh, and contains no provisions for mesh creation. FieldModel is designed for the transfer and interpretation of field data, rather than the grid itself. GrAL is a library for storing, querying and combining meshes, but not for generating them. Additionally, it has only been demonstrated for 2D meshing. GTS is a library of triangular unstructured surface mesh creation and manipulation tools. The target application for this library appears to be the rendering of surfaces, and no volume mesh support is indicated. GGTK is a grid and geometry library with an API of specific grid generation methods for 1-D, 2-D and 3-D meshes. Due to the specific nature of the API, new grid methods would need to be implemented as new API calls, a fact that makes extensibility of the software difficult to maintain. In addition, its requirement of shared topology between grid and geometry is too restrictive for this effort. The TSTT effort, on the other hand, appears to address many of the design goals of this effort, though many aspects of the design (such as mesh generation) are not yet addressed or implemented. However, TSTT is part of a much larger effort (CCA), which limits its flexibility. In addition, TSTT may be too general for this effort’s target audience. Its use of an interface definition language makes it complex to the point of requiring more effort than potential API V2 developers would be willing to invest.

3.5.2 Alternate Design Formats

The most obvious design format alternative to API V1 would be to use an object oriented approach. Based on the programming languages used in the efforts reviewed, this would require the use of C++ as the API environment. Along with the benefits of an object-oriented structure, a C++ API would allow tools such as STL and generic programming to be used for library development.

3.5.3 Industry Standards

Review of the existing efforts indicates that there does not appear to be a standard programming language for analysis APIs. In fact, of the sixteen efforts reviewed, five were written in or supported Fortran77 and Fortran90, seven supported C, and thirteen supported 13 C++. While C++ is arguably the most modern language in general use, a large number of existing analysis applications are written on Fortran variants, due to legacy and efficiency considerations. These potential customers will need to be considered when selecting the API language.

In addition, five APIs were based on language independent interfaces such as IDL, SIDL and CORBA. While a language independent interface offers an API with potential use to nearly *all* applications, it comes with significant overhead. Substantial support software would need to be delivered with such an interface, and proper usage of the API and support software would require a non-trivial training period. Overhead of this magnitude would likely reduce the APIs acceptance and short-term viability in the industry.

Three of the efforts reviewed utilized the C++ concept of Standard Template Libraries. Data structures and algorithms that this generic library provides could be quite useful for the storage and manipulation of data.

3.5.4 Existing Components for Reuse

Several of the efforts reviewed would be good candidates for demonstration of the resulting API V2. In these cases, functions in the existing libraries would be wrapped by API-compliant functions. In order to create a fully compliant API V2 example, it may even be necessary to link several of the libraries simultaneously. Alternatively, where source code is available, the library of API V2 functions could be written based on pertinent functions extracted directly from the code.

All five geometry APIs examined could be used as the foundation of the API-compliant geometry functions. OPEN Cascade would be the leading candidate because of its freeware status, but it could be possible to get a temporary no-cost license of the others for the purpose of demonstrating its use within an API V2 library.

At least three of meshing efforts could also serve as the foundation of an API V2 meshing library. GTS is public domain software with a limited set of unstructured grid creation functions. GrAl source code is also freely available, and contains a number of general-purpose tools for querying and merging mesh elements. As another example, the Mesqute component of the TSTT effort could be used within an API V2 library for mesh quality assessment.

3.5.5 Collaboration

As described above, the TSTT project is the only one even vaguely similar to the API V2 design effort. At first glance, in fact, it appears that it may be possible to develop a high-level grid generation interface common to each effort. This could be done either by making API V2 compliant with TSTT standards or by writing TSTT wrappers around the

API. Though the logistics of maintaining a common API would be difficult to coordinate (different funding agencies, scopes, design goals, etc), it is nevertheless a good idea to open a dialog with the TSTT development team. Specifically, Air Force permission to share information of this effort with TSTT developers should be requested.   This could amount to cross-participation in meetings/teleconferences, sharing of data via password protected web sites, and/or one-on-one visits.

4 Review of UGC API Version 1.0

The first generation of this API was developed through considerable time and effort by members of the Unstructured Grid Consortium. It resulted in a functional programming interface for 1, 2 and 3-D meshes. Despite the success, the consortium understood the need for further work on the project, and asked the Air Force to commission a follow-on development effort.

In order to insure that the original intent and strongest technical features of the API are preserved in the design of the second generation, it was necessary to start with a detailed review of API V1. The beginning sections of this chapter identify and present notable features and shortcomings of API V1 based on different technical standpoints. The final section contains a summary of those ideas and techniques of API V1 that will need to be preserved in API V2.

4.1 Terminology

A number of terms are used repeatedly in the description of API V1.

- **Application**: Program that uses the API to call UGC compliant modules.
- **Modul*e:*** Subroutine or collection of subroutines that satisfies a single API requirement and conforms to UGC programming rules and API specifications. A module may call other modules.
- **Specification:** A list of rules that each module and call in the API must follow.
- **API:** A collection of well-defined calls including the routine name, and argument list to standard UGC functions.
- **Derivative application:** A program that uses UGC API compliant modules to undertake tasks in its application.
- **Application developer:** Makes use of a UGC API compliant module(s) in their programs.

- **Module developer:** Constructs, designs and implements UGC API compliant module code.

4.2 Objectives

The objective of the UGC Version 1.0 API is to define an API for the interoperability of mesh-generation frameworks and modules. The API provides the rules and definitions for interfaces and is intended to encourage the interchange of modules between applications. The API is intended to work with legacy frameworks (Fortran and C) with minimal impact to existing code. The API is designed to be simple to implement, so that the learning curve is minimized. Lastly, API V1 does not specify a geometry package; it will work with the geometry library chosen by the application.

4.3 Scope

The scope of the UGC Version 1.0 API covers both creation and modification of meshes, including surface and volume mesh generation, enhancement of existing meshes, partitioning of meshes, and the computation of metrics to allow the determination of mesh quality. Mesh elements supported include triangles, quadrilaterals, tetrahedra, prisms, and hexahedra. Both new and legacy code are supported and different API-compliant modules may be used concurrently. The API aims to provide an intermediate-level interface focused on large-scale operations; it supports the generation of surface or volume meshes but not the creation of individual mesh nodes. The supporting geometry API called by the parent application is intended to provide essential low-level information, such as surface normal vector calculations. The API does not support structured, overlapping, or hierarchical (Cartesian) meshes.

The existing API V1represents a reduction in scope from its preliminary design. The original plans for API V1 included provisions for geometry modification and repair, as well as additional mesh generation functionality such as quality improvement, joining and merging, transformation and adaption, mesh format conversion, and support for mesh types such as structured, overlapping and Cartesian meshes.

4.4 Architecture

API V1 encompasses mesh generation by dividing the operations into a series of modules. Each module contains the definition of a single API call, and interaction between the application and modules may only take place through API calls. Independent modules representing the same API function are differentiated by a vendor-specific identifier appended to the module name. Additional parameters required by specific modules are created in a separate, API-defined store and are accessed independently of the main API call to the module. Every API function returns a decipherable status flag.

There are several advantages to this type of architecture. First, there is a single standard interface. The API also permits modules to be swapped in and out of an application and shared between multiple applications. The module naming convention will also allow multiple implementations of a module to be utilized within a single process. The calling application can manage the multiple instances of a module with any approach it chooses. The API will be easy to implement if the modules are uniquely named. Lastly, the provision of a separate store for additional parameters will allow extra information to be passed without requiring alteration of the function argument lists.

There are also disadvantages to this approach: When implementing or replacing a module, the API requirement for unique module names means that the application will require code alteration, rather than a mere substitution of the new library. The approach also provides no mechanism for a module to communicate what API calls it users except through documentation. A method is needed to "register" the calls used by a library so that the application can verify that it meets the requirements. The extra store for additional information provides an easy way to circumvent the API structure. It may also place additional code development burden on the application developer when changing modules. Additionally, since the API calls must encompass a large number of approaches, the argument lists to the functions may be very long. Lastly, it is difficult to envision all potential uses of the API and, hence, include all needed functionality into the API specification.

4.5 Data Structure

For data stored within the API specification, one- and two-dimensional arrays of standard data types (character, real, integer and double) are the most complex data type permitted. The defined UGC argument list for the modules is adequate for passing all general mesh data typically required. Additional parameters needed by specific modules are created in a separate store as previously mentioned.

The use of a single common argument list for a module type minimizes code alterations and possible errors when replacing one module with another. The separate store, then, provides the mechanism for passing extra data not encompassed by the argument lists.

A downside to this data structure is that applications not using the standard data format will have to convert data to and from the API format. Also, since every implementation of a module will have different requirements, many optional arguments will have to be populated to get the full functionality of a module. This will limit the plug-and-play benefit of the API. Finally, alteration of legacy code will be required to conform to the API.

4.6 Memory Management

Under API V1, no dynamic resizing of arrays passed in as arguments is allowed inside a module. For modules written in Fortran77, inadequately sized arrays necessarily cause program termination. For languages permitting dynamic memory allocation, the module can be written to return a quantitative request for additional memory. The concept of work arrays is not within the scope of API V1. The only advantages of this approach are that Fortran77 is covered, and that the application controls all accessed memory. Possible downsides are the additional overhead of multiple function calls (to size array requirements and then to allocate memory), the storage of multiple copies of data (one in application, another in the module), and the severe coding restrictions placed on C, C++, and Fortran 90 applications.

4.7 Fortran/C Interface

Fortran and C interfaces are differentiated in API V1 by appending "_F" or "_C" to function names. Wrappers provide language-specific interfaces to the functions. Non-trivial macros are employed to help in data conversion, particularly the handling of strings. As mentioned earlier, 1D and 2D arrays of standard data types are the most complex data type permitted.

This method has the advantage of supporting Fortran77, while still achieving basic required functionality when using higher-level languages. It also requires only one version of the API; the argument lists are identical for Fortran and C. Lastly, the wrappers for each API call only need to be developed once, thereby relinquishing application and module developers from that responsibility.

Unfortunately, this approach severely restricts the data types that may be used when using modern programming languages such as C, C++ and Fortran 90. In addition, wrapper implementation may be non-trivial and may differ among operating systems and compilers. The approach also requires that a library of support tools be linked into the application, and this library may need modification for different compilers and operating systems.

4.8 Error/Message Handling

Errors are returned in API V1 via a mixture of API- and module-specific identification codes. The decoded status value identifies both the API function returning the error and the specific nature of the error. Though specific codes are reserved for common error scenarios (e.g., out of memory), the API can also return custom error codes. This implies that documentation for each module must be provided.

This method provides a means of identifying where an error occurred, but it does not provide module status during execution and does not allow the interruption of an ongoing process. Further, the compressed format of the error codes (less than zero return warning, greater than zero returns error) is non-standard.

4.9 Geometry

API V1 specifies and requires a small set of routinely used geometry calls. For these calls, a geometry identification key is passed to the module, which then extracts the pertinent information. No geometric data is passed as input via the API. The geometry API calls are very loosely coupled to the geometry library itself, with all geometric calculations performed within the geometry library.

This method is general enough to be both useable and easy to implement. The lack of required explicit geometry data keeps function argument lists short. However, the limited subset of geometry routines may not satisfy all needs. In particular, the layer of geometry abstraction and the inability of grid modules to access the geometry may preclude modules that modify geometry. With this method, two integer values are used to represent a geometric entity. Since the usage of these values is not defined, swapping of geometry modules in an application may take significant coding. Furthermore, this method does not cover meshing operations spanning multiple geometry entities. There is also no way to access available topological information, no API call defined for interrogation of geometry quality, and no mechanism for the module to register which geometry calls it requires, nor for the application to verify that it meets all of these requirements.

4.10 Reuse

The review of API V1 above makes it possible to identify a host of features and properties that warrant serious consideration for reuse in API V2.

Objectives*:* The original objectives of API V1 were developed by a team of industry and government organizations, and these objectives were later reiterated by the government sponsor. As the team grows, however, it is possible that new objectives will need to be added. One potential objective may be to provide a design that would easily allow commercial software vendors to develop and market modules. Another may be to align with other standards efforts.

Scope: The scope of API V2 should include API V1's scope as a subset, and should also include a vision of what the final scope will be. API V1 is designed at a very high level. and should be examined to determine if the level is appropriate for V2. In addition, a number of new functions should be defined, including support for low-level functions (accessed many times), interaction with other disciplines (grid adaption, moving meshes, etc.) and optimization packages. Even if the scope is not expanded in API V2, the new API should be designed for extensibility to the envisioned final product.

Further consideration should also be given in API V2 or beyond to a few other items. While API V1 provides a good kernel of functions for geometry handling, more complex examples need to be developed to ascertain the geometric function's true suitability. A set of test scenarios for various commercial grade mesh problems should be used to identify and help detect holes in the API.

Architecture: The overall architecture of API V1 (i.e., application separated from modules through an API) is sound. However, methods for allowing multiple instances of a module that don't require functional name changes should be investigated. Also worth exploring is the concept of data passing between application and modules via a procedural interface rather than through complex single functions with heavily loaded argument lists. Finally, consideration should be given to develop a way for a module to communicate its requirements to the application, similar to the way CCA components register their requirements to a central database.

Data Structure: API V1's data structure is a reasonable choice for cell based data formats, but consideration for edge and face based formats is also warranted. Also, to reduce API parameter lists, ways to allow the data structure to be wrapped up in a C class or Fortran 90 modules should be explored.

Memory Management: Support for FORTRAN77 code severely restricts memory management. The technique of creating large work arrays for memory control helps somewhat, but still requires unnecessary machinations of software. Dynamic memory allocation will almost certainly be a requirement of API V2

Fortran/C Interface: Despite the limitations imposed by FORTRAN77, the ability to mix modules and applications written in Fortran and C via wrapper libraries was demonstrated successfully in API V1. As neither of these programming languages is likely to diminish in importance in the next several years, this dual language support should continue to be maintained.

Geometry: The geometry functions designed in API V1referred to curve and surfaces by an array of 2 integer values. This ID approach simplified these functions' argument list, thereby reducing the effort to use them within the application and other modules. All of the geometry maintenance became the responsibility of the modules defining the functions. This same ID method should probably be extended to API V2. Surface meshing will need to be extended to more than one surface at a time, however.

5 Recommendations for API Version 2.0

In the previous chapter, several features of API V1 were identified as strengths, with several more identified as weaknesses. This information in combination with the knowledge obtained from the related effort review makes it possible to offer recommendations prior to the design and development of API V2.

5.1 API Namespace and Library Management

The vendor specific identifier in the API prototypes forces application developers to modify source calls for all function calls when switching from one API-compliant library to another. In order to eliminate this inconvenience, it is recommended that this vendor-specific identifier be eliminated, changing function calls from **`UGC_modulename_uniqueIdentifier{F|C}()`** to **`UGC_modulename().`** Vendor information for a given UGC library could be passed using a **`UGC_getVendorString()`** type API call.

It is the application's responsibility to insure that the proper API V2-compliant library is linked to the application. In some scenarios, an application may need to access multiple API-compliant libraries, either serially or simultaneously. This necessitates a dynamic library loading capability within the application. One such library for exactly that purpose is the GNU-licensed software known as GLib/GModule [23].

Though software for dynamic library management will not be provided in the distribution, functions to facilitate such management should be included in the API. The GLib library is a logical choice for testing, certifying and demonstrating these specialized yet important API functions.

5.2 API Programming Language

API V1 specified both Fortran and C representations of the API. It also provided header files (**`UGC_bind_f_and_c`**) that allowed Fortran applications to link with C libraries, and vice-versa. Though this satisfied users of mixed language applications, when coupled with the fact that all data resided on the application side (see Section 5.3), it resulted in unwieldy argument lists. This was further exacerbated by API V1's support for Fortran77, a language without dynamic memory allocation. Support for Fortran77 forced high-level API functions to be applied twice – once to determine the amount of memory required to perform an operation, and a second time after the memory had been allocated on the application side. When the application itself was written in Fortran77, recompilation was often necessary to accommodate the sizing demands of the particular problem. These problems in combination made the usability of API V1 more a computer science than a grid generation issue, which was obviously not the original intent.

Therefore, it is recommended that C be the programming language of API V2. This will allow prototypes to be leaner, more easily understood and more pertinent to grid generation issues. Though meshing libraries adhering to the API standard will be written in C, consideration for those legacy and new applications written in other languages such as Fortran77 and Fortran90 will still be required. To encourage this new API's usage among Fortran application developers, it may also be prudent to offer source code to C/Fortran wrapper functions that allow the API to be called from within Fortran applications. Converse to this situation is the development of an API-compliant library whose baseline language is Fortran. In this case, mesh library developers will need to write Fortran/C wrappers that will allow the library to be API-compliant. If time permits, it may be possible to deliver source code (as an example or a complete wrapper library) for this purpose.

The final API definition should be delivered via a series of C-style header files containing API prototypes, published data structures, typedefs, macros and atomic data types (e.g., UGC_Int).

5.3 Data Passing

API V1uses a small number of high level functions such as **UGC_MeshVolume()**. In this particular example, the entire volume mesh generation process is performed within a single API call. This includes passing all of the boundary data into the function, and returning all of the generated cell data back from the functions. The sheer amount of data transfer required for such an involved operation results in very lengthy and complex argument lists. All meshing applications are required to convert input data into the published formats, and later to convert output data into the application-specific format. In general, a meshing library will not have a need for all data returned from the function, but will need to receive it nevertheless, resulting in unneeded work.

In API V2, the entirety of mesh data will not be required to be input and output with every API call, but rather, will reside within the API-compliant library. Applications will then be able to query and retrieve fine-grained mesh data on an as-needed basis. This will make data transfer significantly easier to manage, possibly allowing data to be retrieved directly in a format required by the receiving application. An example usage of such an API might be as follows:

```
UGC_Mesh_Surface(input1,input2,...,(void **)&MeshPtr );
UGC_InquireNodes((void *)MeshPtr, (int *)&numNodes,(structure NodeData **)Nodes );
UGC_InquireCells((void *)MeshPtr, (int *)&numCells,(structure CellData **)&Cells );
UGC_Free_Ptr((void **)&MeshPtr);
```

In the first API call, the surface mesh is generated. Next, the node values in the mesh are returned. After this the mesh cells are returned, and finally, the mesh data is released from memory on the library side. Querying functions at an even lower level than this example (such as individual nodes and cells) might also be warranted.

A demonstration of this low-level procedural data interface should also be provided. One potential application would create a simple mesh and use querying-style API calls to populate an existing grid format such as GrAL or AOMD.

5.4 Geometry

Generally there are only a few basic geometric functions that are required by most meshing applications. These functions, however, tend to require considerable infrastructure, and are often developed using more-general geometry libraries, either commercial or public domain. Since many potential meshing libraries will be unable to deliver the geometry engine with the library, the geometry functionality will be the responsibility of the application. This will allow developers to deliver general meshing libraries without having to provide the underlying geometry tools. Broadcast of the application's geometry functions to the meshing library will be handled via callback function registration.

```
UGC_RegisterCallBack_GeomEval( surfeval ) ;
UGC_RegisterCallBack_GeomProj( surfproj ) ;
```

In this example, **`surfeval`** and **`surfproj`** are pointers to application-side functions. The geometry callback functions should not require the entire geometry definition to be included of its argument list. Rather, entities should be passed by opaque references or IDs. IDs can be de-referenced to integer values or pointers, but it will be up to the geometry functions within the application to determine the proper context.

It will also be necessary to select a geometry engine to be used for demonstration of these features. Any number of engines should suffice, including OPEN Cascade, CAPRI, and proprietary APIs used at Pointwise, Boeing and Lockheed. Though no source code will be delivered in this demonstration, OPEN Cascade is particularly attractive because it is open source, and can be shared among companies involved in the demonstration.

5.5 Optional Data

Two types of optional data are needed in API V2. The first of these provides for optional functions and operations on mesh data, while the second pertains to tunable parameter data used to control the meshing process.

While API V1 addressed the most prominent requirements of meshing, namely curve, surface and volume mesh generation, it did not address peripheral issues such as mesh quality assessment or adaption, for example. Handles to generic ancillary meshing functions will need to be added to the API for those libraries that provide them. Knowledge of these functions should be provided via querying functions in the library that return the number of optional functions, a character-string descriptor of each function, and a text-based description (including function prototypes) of each function. Execution of these optional functions can then be controlled via a generic execution function that takes the descriptor as an argument.

Another type of function desired by some meshing libraries might be a user-provided low-level function used in a higher-level grid function. For example, certain meshing algorithms require background meshes to provide target cell sizes at a point in physical space. While any API V2 compliant library using such a feature would need to provide a function for this purpose, the application may wish to provide an alternative algorithm. For these types of functions, the alternative algorithm should be communicated to the meshing library via callback registrations such as **`UGC_RegisterCallBack_BackgroundEval(myFunction).`** Again, prototypes for these special functions will need to be published for the application developer.

The second type of optional data is tunable parameter data. This corresponds to the user-controlled input parameters to specific meshing algorithms. Though tunable parameters reduce meshing library portability, they are required in instances where non-standard library parameters significantly influence mesh behavior. These optional data functions will be based on the UGC Management functions of API V1. Default values for all parameter values should be assigned automatically, and should be the mesh library's responsibility to insure that API functions work for the majority of cases with the default values.

All tunable parameters available in the particular meshing library should be published via an API function call such as **UGC_GetAllOptionNames().** The meshing library should register parameters and defaults via calls such as
UGC_SetIntOption("Param1",100,"This sets the Parameter1 value").
The meshing application should in turn be able to overwrite the default tunable values with calls such as **UGC_SetIntOption("Param1",200,NULL).**
Current values of tunable parameters should be queried from the meshing application with calls like **UGC_GetIntOption("Param1",&Value,Description)**,

which will return the current parameter value along with a short description. Consideration for a function providing a usage description should also be given, such as **UGC_GetUsageOption("Param1",UsageDescription).** This optional-data framework allows all options to be presentable in a GUI-based environment, allowing users to select pertinent variables to be used.

6 Coordination With Other Efforts

During the course of the API development effort, several initiatives were undertaken to coordinate with other organizations. These initiatives were designed to coordinate the API development effort with similar activities taking place at the Department of Energy and NASA Langley. An effort was also made to maintain communication with developers of API V1 and with the Unstructured Grid Consortium (UGC) steering committee.

Periodically throughout the development effort, a status update was presented to the UGC. These briefings were given at the 2003 and 2004 annual meetings as well as several teleconferences. These meetings were an excellent opportunity to present the API progress and seek feedback from UGC members. In addition, access to the GGMNS development web site was provided to the UGC steering committee. A special teleconference was held with Bill Jones from NASA Langley to collect his feedback. Bill was a member of the API V1 design team and has been a supporter of the UGC API concept. Several API design decisions were made based on the feedback from these important interactions.

Early in the GGMNS design effort, Lori Freitag Diachin of the Department of Energy TSTT program was contacted and the TSTT and UGC API efforts were discussed. Differences in approaches prevented direct coordination of the two efforts. The progress of the TSTT development was monitored throughout the UGC API development. Members of the GGMNS team joined the TSTT mailing list, and access to the GGMNS development web site was provided to the TSTT team.

7 API Version 2.0 Design

The design of API V2 is based on a number of considerations outlined during the review phase as well as a number discovered during the design process. This chapter discusses the scope of the API, conventions used in the API formatting, terminology used in the design, the basic layout of the API library, and several different ways in which the API might be used.

7.1 Scope

A number of factors define the scope of API V2, including the API programming language, the types of meshes and element types to be supported, the types of supported data representation, and any known limitations of the API. Each of these issues is addressed below.

7.1.1 API Programming Language

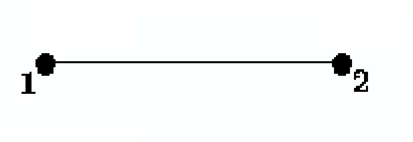
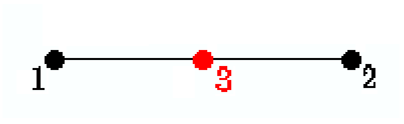
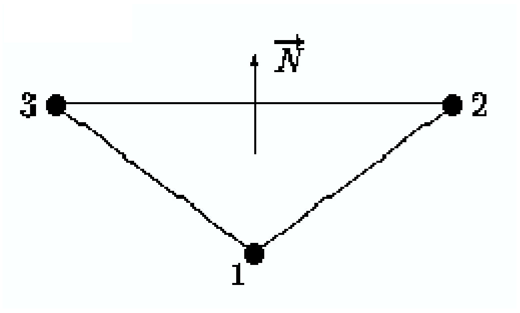
The UGC API V2 will specify C as the standard programming language. Other programming languages such as Fortran77 and Fortran90 will be supported via wrappers. It is up to the application and/or library developer to develop these wrappers. Example wrappers have been provided (See Chapter 9).

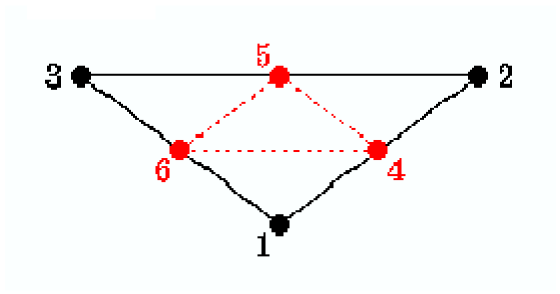
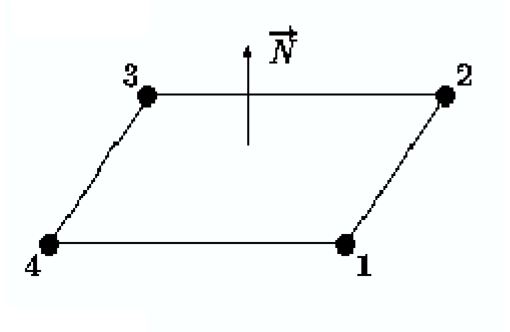
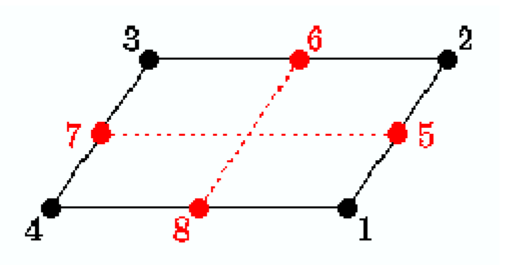
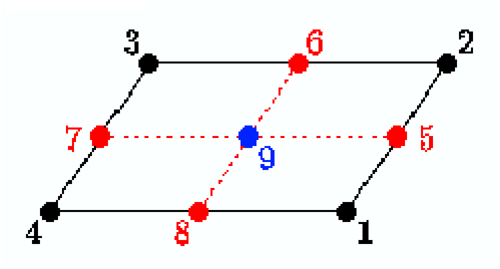
7.1.2 Mesh Types

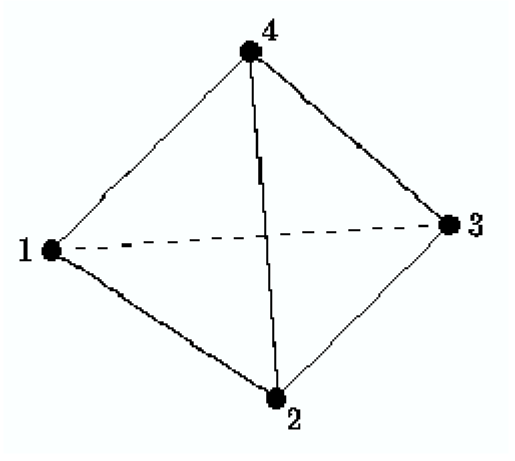
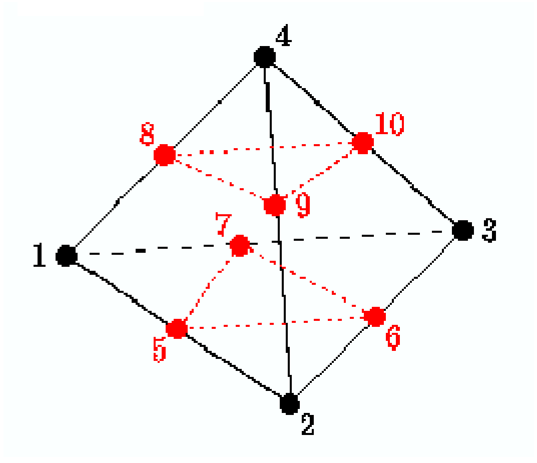
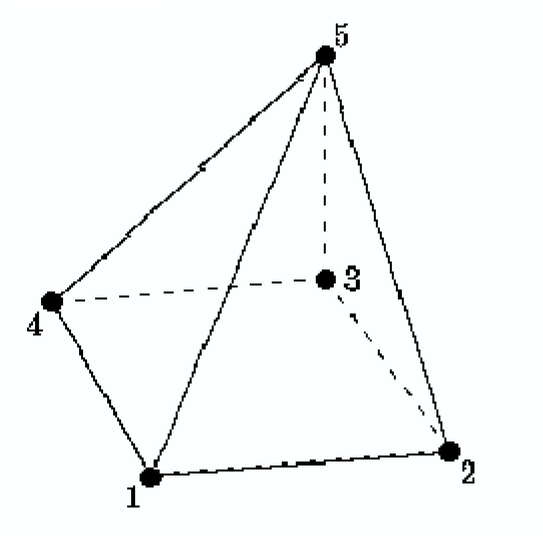
The UGC API V2 will support mesh generation for 1-D, 2-D, and 3-D unstructured finite element type meshes.

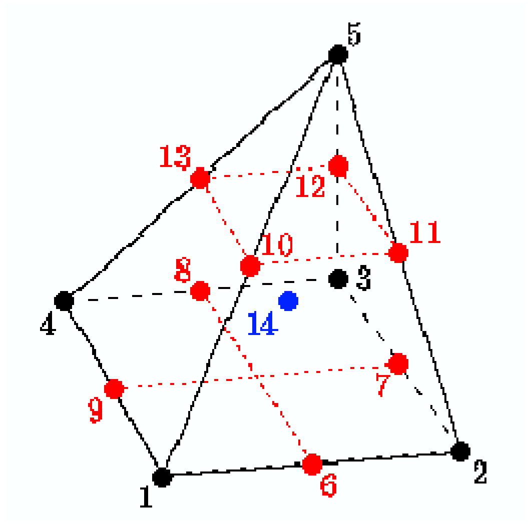
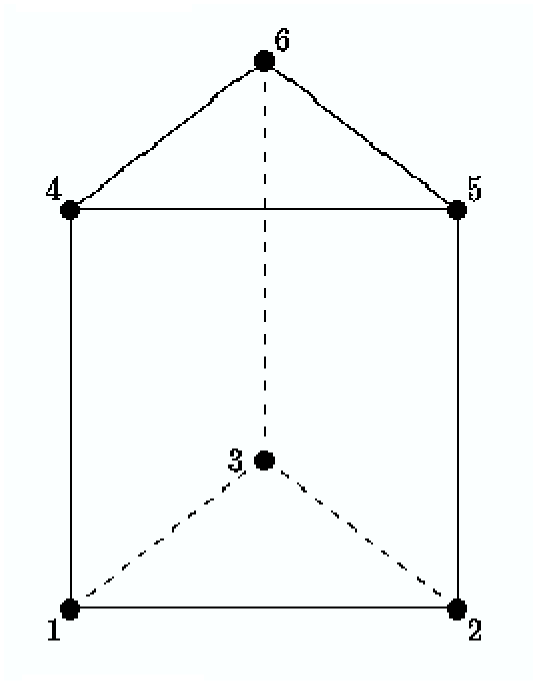
7.1.3 Element Types

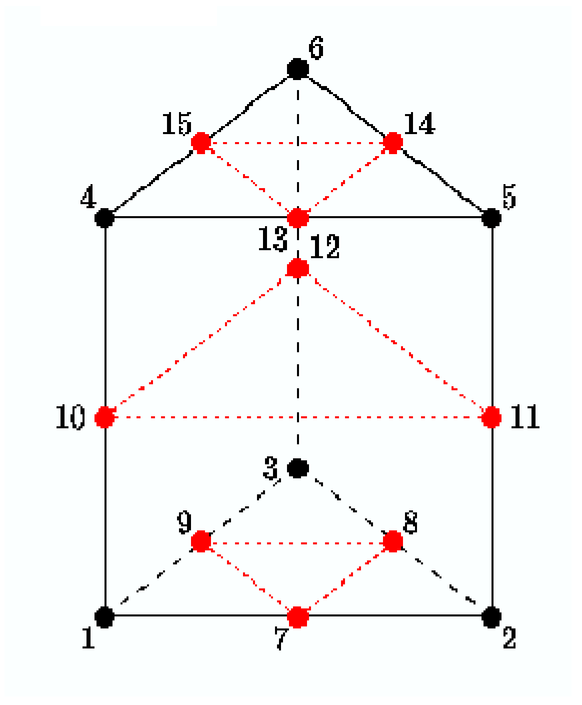
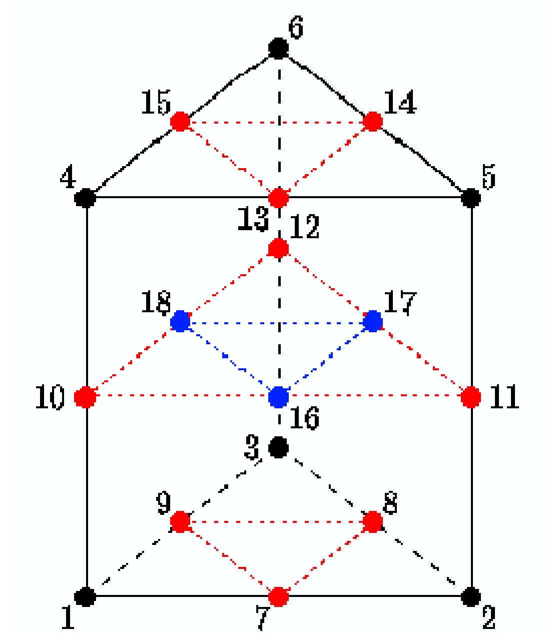
The UGC API V2 supports mesh elements that are: linear, triangular, quadrilateral, tetrahedral, pyramid, pentahedral (prism), and hexahedral. The elements may have edges and faces that are either linear or quadrilateral. Node numbering and arrangement of the supported elements are detailed in Table 1.

Element Type and Description	**ID #**	**Node Schematic**
`UGCDB_ELEMTYPE_BAR_2` Linear Edge	**1**	
`UGCDB_ELEMTYPE_BAR_3` Quadratic Edge	**2**	
`UGCDB_ELEMTYPE_TRI_3` Triangle with Linear Edges	**10**	

UGCDB_ELEMTYPE_TRI_6 Triangle with Quadratic Edges	**11**	
UGCDB_ELEMTYPE_QUAD_4 Quadrilateral with Linear Edges	**12**	
UGCDB_ELEMTYPE_QUAD_8 Quadrilateral with Quadratic Edges and No Interior Nodes	**13**	
UGCDB_ELEMTYPE_QUAD_9 Quadrilateral with Quadratic Edges and Interior Nodes	**14**	

UGCDB_ELEMTYPE_TETRA_4 Tetrahedral with Linear Edges	**20**	
UGCDB_ELEMTYPE_TETRA_10 Tetrahedral with Quadratic Edges	**21**	
UGCDB_ELEMTYPE_PYRA_5 Pyramid with Linear Edges	**22**	

UGCDB_ELEMTYPE_PYRA_14 Pyramid with Quadratic Edges	**23**	 5 13 12 11 10 8 3 4 14 9 7 2 1 6
UGCDB_ELEMTYPE_PENTA_6 Pentahedron with Linear Edges	**24**	

UGCDB_ELEMTYPE_PENTA_15 Pentahedron with Quadratic Edges and No Interior Nodes	**25**	 6 15 14 4 5 13 12 10 11 3 9 8 1 7 2
UGCDB_ELEMTYPE_PENTA_18 Pentahedron with Quadratic Edges and Interior Nodes	**26**	 6 15 14 4 5 13 12 18 17 10 11 16 3 9 8 1 7 2

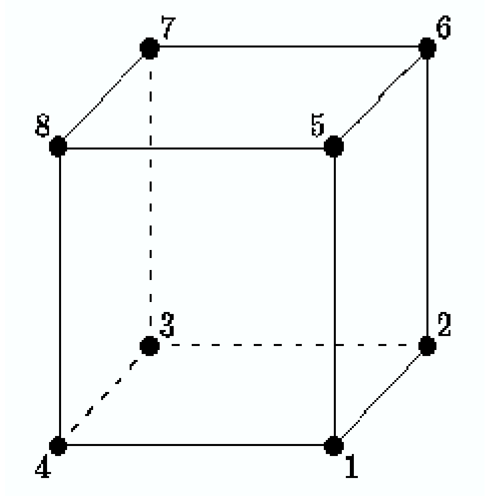
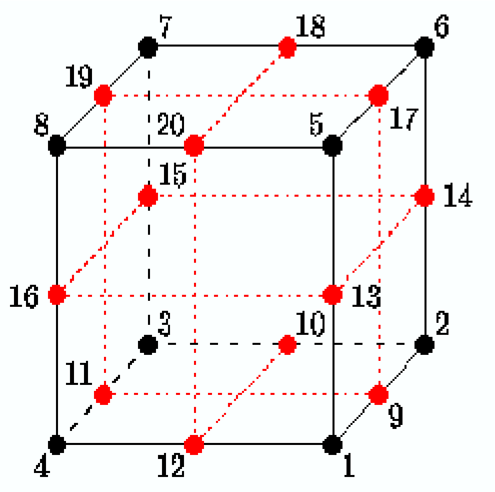
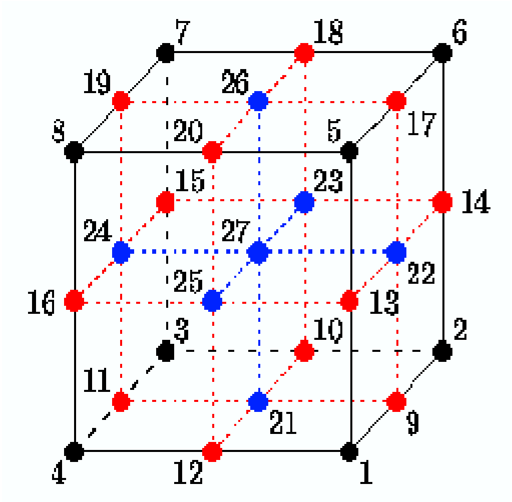
UGCDB_ELEMTYPE_HEXA_8 Hexahedron with Linear Edges	**27**	
UGCDB_ELEMTYPE_HEXA_20 Hexahedron with Quadratic Edges and No Interior Nodes	**28**	 7 18 6 19 17 8 20 5 15 14 16 13 3 10 2 11 9 4 12 1
UGCDB_ELEMTYPE_HEXA_27 Hexahedron with Quadratic Edges and Interior Nodes	29	 7 18 6 19 26 17 8 20 5 15 23 14 24 27 22 25 16 13 3 10 2 11 21 9 4 12 1

Table 1. Supported Element Types

7.1.4 Atomic Data Types

The UGC API V2 defines atomic data types for integers, reals, characters and Booleans. The details of the definitions are outlined below in Table 2.

Atomic Data Type	typedef	Description
`UGC_Int`	`long`	Atomic integer; can be 32- or 64-bit
`UGC_Real`	`double`	Atomic floating point number; must be 64-bit
`UGC_Char`	`unsigned char`	Atomic 1-byte character; must be 8-bit

`UGC_Boolean`	`int`	Boolean - Has two possible values:
		`#define UGC_FALSE (0)`
		`#define UGC_TRUE (!UGC_FALSE)`

Table 2. Atomic Data Types

7.1.5 Assumptions and Limitations

Restrictions and assumptions pertaining to geometry, topology, mesh and mesh geometry elements are described in Section 7.3.

The topological structure of a mesh model constructed by the API functions of Section 7.4 must remain constant for the life of the model. This means that modification to the topology may not be performed with API functions directly. Instead, the application must make a copy of the lower-order topology and mesh elements and build a new topological structure. An example of a modified mesh topology is a surface mesh that has a hole added to it after perhaps an intersection with another surface. This particular action increases the number of loops in the sheet by one, which constitutes a topology change.

7.2 Conventions

Conventions for naming functions and error codes are described in this section.

All functions within the UGC API V2 are named according to a convention so that the function names are easy to understand. All function names are of the form: **`<prefix>_<object>_<verb>_<object/qualifiers>.`** A list of the currently implemented values for the terms contained within the angled brackets (< >) is contained in Table 1.

Term	Values	Descriptions

prefix	**UGC**	**General UGC Function**
	UGCDB	**UGC Database**
	UGCG	**UGC Geometry**
	UGCM	**UGC Mesh Function**
	UGCPD	**UGC Parameter**
	UGCP	**UGC Plugin Function**
object	**Session**	**N/A**
	Entity	**N/A**
	Model	**N/A**
	Volume	**N/A**
	Shell	**N/A**
	Sheet	**N/A**
	Loop	**N/A**
	String	**N/A**
	Cell	**N/A**
	Face	**N/A**
	Edge	**N/A**
	Vertex	**N/A**
	SurfaceVertex	**N/A**
	CurveVertex	**N/A**
	Algorithm	**N/A**
	Parameter	N/A
	AlgorithmProperties	**N/A**
	ParameterProperties	**N/A**

verb	Create	N/A
	Inquire	N/A
	Delete	N/A
	Attach	N/A
	Assign	N/A
	Use	N/A
	Register	N/A
	Destroy	N/A
	Execute	N/A
	Remove	N/A
	Load	N/A
	Unload	N/A
object/qualifiers	Model	N/A
	Class	N/A
	Volume	N/A
	Volumes	N/A
	Sheet	N/A
	Sheets	N/A
	String	N/A
	Strings	N/A
	Loop	N/A
	Loops	N/A
	Vertices	N/A
	Vertex	N/A

	SurfaceVertices	N/A
	Cells	N/A
	Surface	N/A
	Faces	N/A
	Curve	N/A
	CurveVertices	N/A
	Edges	N/A
	ElemType	N/A
	Point	N/A
	TopologicalType	N/A
	AnySurfaceVertex In Sheet	N/A
	SurfaceVertices In Sheet	N/A
	AnyCurveVertex In String	N/A
	CurveVertices In String	N/A
	SurfaceGeometry	N/A
	CurveGeometry	N/A
	StatusDescription	N/A
	Algorithm	N/A
	Algorithms	N/A
	DirectMeshing	N/A
	AlternatePlugin	N/A
	Properties	N/A
	Parameters	N/A
	Key	N/A

	Keys	**N/A**
	ValueFromKey	**N/A**
	ValueToKey	N/A
	Plugin	**N/A**

Table 3. Namespace Terms

A list of the error codes implemented in UGC API V2 is presented below in Table 4.

Error Code	Value	Description
UGCDB_STATUS_NOERRORS	**0**	UGC Database Function: No Errors Occurred
UGCDB_STATUS_GENERALERROR	**1**	UGC Database Function: Unspecified Error
UGCDB_STATUS_MEMORYALLOCATIONERROR	**10**	UGC Database Function: Memory Allocation Error
UGCG_STATUS_NOERRORS	**0**	UGC Geometry Function: No Errors Occurred
UGCG_STATUS_APPLICATION_BASE	**32768**	UGC Geometry Function: Application Specific Error Codes Start at this Value
UGCM_STATUS_NOERRORS	**0**	UGC Mesh Function: No Errors Occurred
UGCM_STATUS_GENERALERROR	**1**	UGC Mesh Function: Unspecified Error
UGCM_STATUS_APPLICATION_BASE	**32768**	UGC Mesh Function: Application Specific Error Codes Start at this Value
UGCPD_STATUS_NOERRORS	**0**	UGC Parameter Function:

		No Errors Occurred
`UGCPD_STATUS_GENERALERROR`	**`1`**	UGC Parameter Function: Unspecified Error
`UGCP_STATUS_NOERRORS`	**`0`**	UGC Plugin Function: No Errors Occurred
`UGCP_STATUS_GENERALERROR`	**`1`**	UGC Plugin Function: Unspecified Error

Table 4. Error Codes

7.3 Terminology

The UGC API V2 approaches generalized mesh generation by dividing its universe into four classes of entities. At the top level are the **topology** entities, which establish and maintain the manner in which **mesh** entities are inter-related. Mesh entities, the ultimate product of UGC-compliant libraries, are owned by the topology entity on which they are defined. At the lowest level are **geometry** entities, which provide the shape to which individual mesh entities will be constrained. The relationships between mesh and geometry entities, when they exist, are maintained via the **mesh geometry** entities. The UGC entity hierarchy is illustrated in Figure 1.

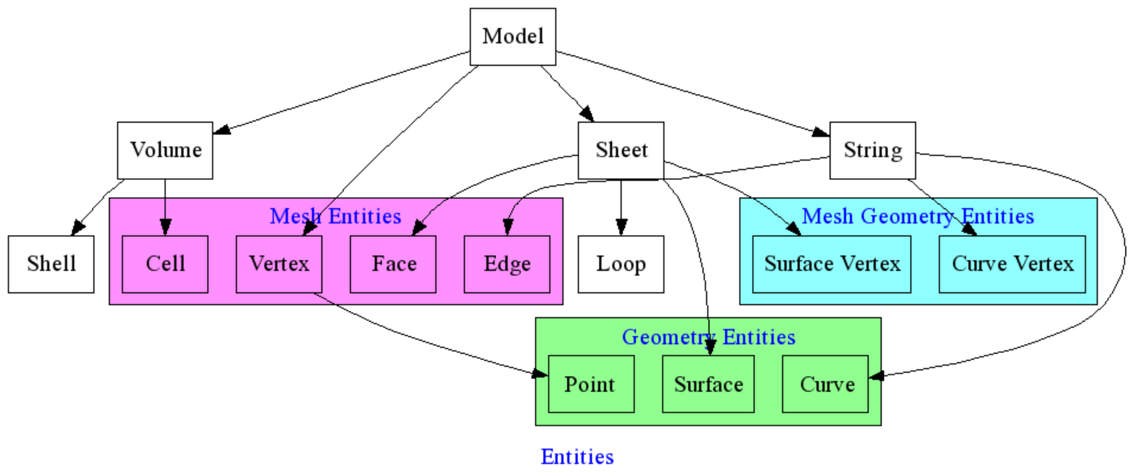


Figure 1. UGC API V2 Entity Hierarchy

7.3.1 Mesh Topology Entities

Topological entities are those entities that provide the framework for the mesh model. They prescribe the manner in which individual mesh components are inter-related, and serve as the parent entities on which mesh entities are defined. Topology entities are arranged in a hierarchical manner, with the model entity at the top, and the string entity at the bottom.

Entity	Description	Bounds	Bounded By	Mesh Elements
Model	framework for creating a connected or logically related set of mesh elements in 1-D, 2-D or 3-D - this is the highest level entity	-	Volumes, Sheets, and Strings	Vertices
Volume	connected subset of 3-D space	Model	Shells	Cells
Shell	unordered list of oriented Sheets	Volume	-	-
Sheet	manifold subset of 3-D space that may be supported by a Surface and is locally deformable to a plane	Model or Volume	Loops	Faces and Surface Vertices
Loop	ordered list of Surface Strings	Sheet	-	-
String	manifold subset of 3-D space that may be supported by a Curve and is locally deformable to a line	Model or Loop	Vertices (implicitly)	Edges and Curve Vertices

Table 5. Topology Entities

7.3.2 Mesh Entities

Mesh entities contain the actual mesh data required by the application. They are linked directly to topology entities following the hierarchy described above. The types of mesh elements allowed are prescribed per algorithm in the meshing library.

Entity	Description	Owned By
Cell	computationally 3-dimensional meshing element	Volume
Face	computationally 2-dimensional meshing element	Sheet
Edge	computationally 1-dimensional meshing element	String

Vertex	computationally 0-dimensional meshing element	Model

Table 6. Mesh Entities

7.3.3 Mesh Geometry Entities

Mesh geometry entities serve as the bridge between geometry and mesh entities. Specifically, they maintain the relationship between a point's representation on an abstract geometric entity and its vertex position in 3-D space. Any number of curve or surface vertices may link to the same mesh vertex (thereby providing connectivity), but a given curve or surface vertex will always link to exactly one mesh vertex.

Entity	Description	Owned By
Surface Vertex	imprint of a Vertex on a Sheet, allowing several surface normals and surface parameters to be associated explicitly with a single Vertex	Sheet
Curve Vertex	imprint of a Vertex on a String, allowing several curve tangents and curve parameters to be associated explicitly with a single Vertex	String

Table 7. Mesh Geometry Entities

7.3.4 Geometry Entities

Three types of geometry entities are supported, corresponding to 0, 1, and 2-D parametric shapes.

Entity	Description	Owned By
Surface	computationally 2-dimensional support geometry - examples of this include: • single biparametric CAD surface • collection of surface mesh elements with no global 2-D parameterization • collection of CAD surfaces with no global 2-D parameterization • any set of geometry that uniquely maps points in 2-D in the parametric vicinity of the mesh to a connected, continuous subset of 3-D space that is	Sheet

	locally mapable to a plane • any set of geometry that uniquely maps points in 3-D in the vicinity of the mesh to a connected, continuous subset of 3-D space that is locally mapable to a plane	
Curve	computationally 1-dimensional support geometry - examples of this include: • single parametric CAD curve • collection of curve mesh elements with no global 1-D parameterization • collection of CAD curves with a global 1-D parameterization • any set of geometry that uniquely maps points in 1-D in the parametric vicinity of the mesh to a connected, continuous subset of 3-D space that is locally mapable to a line • any set of geometry that uniquely maps points in 3-D in the vicinity of the mesh to a connected, continuous subset of 3-D space that is locally mapable to a line	String
Point	computationally 0-dimensional support geometry	Vertex

Table 8. Geometry Entities

7.3.5 Non-Entity Abstractions

A few additional terms that are used frequently in API V2 are defined below.

Entity	Description
Session	lifespan of a library within an execution thread
Algorithm	meshing algorithm that can be executed in the current session
Parameter	key/value pair that can be associated with mesh entities on input to control the execution of an algorithm or on output to reflect computed data

Table 9. Non-Entity Abstractions

7.4 Architecture

One of the primary design goals of API V2 was to reduce the size of the argument lists for API functions. Since the overall amount of data required by a meshing algorithm is not negotiable, reduced argument lists imply more API functions. This in turn suggests that the meshing data reside in library memory, since data will need to be transferred into memory in a piecemeal manner, and transferred back to the application in a likewise manner. This approach differs from API V1 in the respect that API V1 transferred all meshing data in and out of the meshing library within a single API call, essentially allowing the mesh data to reside on the application side. The idea of maintaining the grid within the library reduces the burden normally placed on the application developer, since grid elements can be stored and retrieved individually from the application. This fact removes the need for temporary array construction for the purpose of shuttling all data in and out of the library within a single call. However, the burden is now shifted to the library developer, who will need temporary data structures for the purpose of storing mesh data. It also forces the library developer to write a large number of functions for the purpose of data transfer between the application and the developer.

Another primary design goal of API V2 was to allow for serial or parallel usage of multiple UGC libraries within a single application. However, if the memory for a grid is resident to the library that created it, the data will not be addressable by functions from a different UGC library. Here, the application is forced to transfer library1 data to the application, make a copy of it and transfer the data to Library2. Besides requiring a duplication of grid components between the two libraries, it also requires the application to broker the transfer via repeated usage of the data transfer API functions.

For this reason a two-library architecture was selected. The first library, referred to as the **meshing library,** contains a minimal set of API functions that primarily perform the tasks associated with mesh generation. Because it is a lean set of commands, library developers will be able to concentrate on mesh generation, rather than on populating a large number of data transfer functions. Multiple meshing libraries can exist in an application simultaneously. The second library is referred to as the **database library,** and serves as a central data repository for all mesh data. There will never be more than one database library in a given application.

The database library contains a larger number of functions than the meshing library, most of which are designed for data transfer between the application, the meshing library, and the database library. As stated earlier, development of a database library would take a substantial effort, due to the number of functions required and the creation of data structures for grid data. Nevertheless, the task of writing such a library is fairly mechanical, and likely would not change considerably from one developer’s database library instance to another’s. With this in mind, an example database library was written under this effort, and is to be included in source code form with the UGC API V2 library distribution (see Chapter 9). Direct use of this example database library will allow library developers to create a UGC-compliant library with minimal effort, requiring software implementations of the small set of mesh library functions only.

All of the functions defined in UGC API V2 follow the general prototype

```
extern UGC_Status UGC_<object>_<verb>_<object/qualifiers> (...);
```

where **`<object>`** refers to the type of object on which the function is applied, **`<verb>`** refers to an action taken on the object, and **`<object/qualifier>`** refers to a qualifier on the action.

7.4.1 Database Library

The database library is used for transfer of mesh data, assignment and query of generic parameter data such as mesh library run-time parameters, and assignment of plugin mesh library data. Its API is defined in a total of four header files **`ugc_database.h`**, **`ugc_geometry.h`, `ugc_parameter.h`**, and **`ugc_plugin.h`**), divided for clarity by functional classes.

7.4.1.1 Data Transfer Functions

Transfer of entities to and from the meshing library is governed by strict topological rules defined by the entity hierarchy.

Population of the mesh database begins with the topological entities that glue the mesh elements together. The meshing hierarchy is typically created in a top-down manner, beginning with models, and continuing with volumes, shells, sheets, loops and strings with the following functions:

- **`UGCDB_Session_Create_Model() :`** creates an empty model for meshing purposes.
- **`UGCDB_Model_Create_Volume() :`** creates an empty volume for meshing purposes.
- **`UGCDB_Model_Create_Sheet() :`** creates an empty sheet for meshing purposes.
- **`UGCDB_Model_Create_String() :`** allows the creation of an empty string in a model.
- **`UGCDB_Model_Delete() :`** deletes a model and all its associated data from the session.
- **`UGCDB_Volume_Create_Shell() :`** creates a boundary shell around a volume from an unordered list of oriented sheets.
- **`UGCDB_Sheet_Create_Loop() :`** creates a boundary loop on a sheet from an ordered list of surface strings.

After topology is established, topological entities are tied to geometry with the following commands:

- **`UGCDB_Sheet_Attach_Surface() :`** associates an abstract geometry surface with a sheet.
- **`UGCDB_String_Attach_Curve() :`** associates an abstract geometry curve with a string.

Another group of functions is used to create and assign vertex, face and cell mesh elements to topological model, sheet and string elements, respectively.

- **`UGCDB_Model_Create_Vertices() :`** creates vertices in a model from their coordinates.
- **`UGCDB_Volume_Create_Cells() :`** creates cells in a volume from the ordered list of vertices that bound each cell.
- **`UGCDB_Sheet_Create_Faces() :`** creates faces in a sheet from the ordered list of surface vertices that bound each face.
- **`UGCDB_String_Create_Edges() :`** creates edges in a string from the ordered list of curve vertices that trace each edge.

Similarly, a series of functions exist for the deletion of individual mesh elements.

- **`UGCDB_Cell_Delete() :`** deletes a cell and all its associated data from the owning volume.
- **`UGCDB_Face_Delete() :`** deletes a face and all its associated data from the owning sheet.
- **`UGCDB_Edge_Delete() :`** deletes an edge and all its associated data from the owning string.
- **`UGCDB_Vertex_Delete() :`** deletes a vertex and all its associated data from the owning model.

Mesh geometry entities represent the link between mesh vertices and the geometric entities on which they lie. Creation, deletion, and assignment of these entities to geometry entities is controlled with the functions below:

- **`UGCDB_Sheet_Create_SurfaceVertices() :`** creates surface vertices, associating the surface geometry of a specific sheet with some vertices in a model.
- **`UGCDB_SurfaceVertex_Assign_SurfaceGeometry() :`** assigns the surface geometry associated with a surface vertex, allowing the position of the surface vertex to vary over time while maintaining its identity.

- **`UGCDB_SurfaceVertex_Delete() :`** deletes a surface vertex and all its associated data from the owning sheet.

- **`UGCDB_String_Create_CurveVertices() :`** creates curve vertices, associating the curve geometry of a specific string with some vertices in a model.

- **`UGCDB_CurveVertex_Assign_CurveGeometry() :`** assigns the curve geometry associated with a curve vertex, allowing the position of the curve vertex to vary over time while maintaining its identity.

- **`UGCDB_CurveVertex_Delete() :`** deletes a curve vertex and all its associated data from the owning string.

- **`UGCDB_Vertex_Assign_Point() :`** assigns the geometry associated with a vertex, allowing the position of the vertex to vary over time while maintaining its identity.

Typical usage of the UGC API V2 involves the building of topology and mesh elements using the functions described above. When these entities have been established, the next step is to use the UGC library meshing functions (described later in the section) to create new mesh elements, followed by invoking UGC querying functions for the extraction of newly formed meshing data. A large number of functions have been created for exactly this purpose, that of querying information from existing topology, geometry, mesh geometry and mesh entities. The broad set of functions described below affords the application and library developers full access to data existing in the database library, defined either within the application or within the library.

The first group of inquiry functions contain those used to identify the topology, mesh, and mesh geometry entities attached to a topology entity. The following functions query the topological hierarchy:

- **`UGCDB_Model_Inquire_Volumes() :`** finds volumes in a model.

- **`UGCDB_Model_Inquire_Sheets() :`** finds sheets in a model.

- **`UGCDB_Model_Inquire_Strings() :`** finds strings in a model.

- **`UGCDB_Volume_Inquire_Model() :`** finds the model that owns a volume.

- **`UGCDB_Volume_Inquire_Shells() :`** finds the shells that bound a volume.

- **`UGCDB_Shell_Inquire_Volume() :`** finds the owning volume of a shell.

- **`UGCDB_Shell_Inquire_Sheets() :`** finds the unordered list of oriented sheets in a shell.

- **`UGCDB_Sheet_Inquire_Model() :`** finds the model that owns a sheet.

- **`UGCDB_Sheet_Inquire_Shells() :`** finds the 0, 1, or 2 shells adjacent to a sheet.
- **`UGCDB_Sheet_Inquire_Loops() :`** finds the loops that bound a sheet.
- **`UGCDB_Loop_Inquire_Sheet() :`** finds the owning sheet of a loop.
- **`UGCDB_Loop_Inquire_Strings() :`** finds the ordered list of surface strings in a loop.
- **`UGCDB_String_Inquire_Model() :`** finds the owning model of a string.
- **`UGCDB_String_Inquire_Loops() :`** finds the loops radially adjacent to a string.

Three related functions are used to identify the geometry and mesh geometry entities linked to topological entities.

- **`UGCDB_Sheet_Inquire_Surface() :`** finds an abstract geometry surface associated with a sheet.
- **`UGCDB_Sheet_Inquire_SurfaceVertices() :`** finds the surface geometry of a specific sheet associated with some vertices in a model.
- **`UGCDB_String_Inquire_Curve() :`** finds an abstract geometry curve associated with a string.

When a meshing algorithm is applied to a model, volume, sheet or string, new mesh elements are often created. The functions below are used to retrieve mesh element data from topological entities:

- **`UGCDB_Model_Inquire_Vertices() :`** finds vertices in a model.
- **`UGCDB_Volume_Inquire_Cells()`** : finds the cells that are contained in a volume.
- **`UGCDB_Sheet_Inquire_Faces() :`** finds the faces that are contained in a sheet.
- **`UGCDB_String_Inquire_Edges() :`** finds the edges that are contained in a string.

The second group of inquiry functions contains those used to identify the topology, geometry, and mesh entities belonging to mesh geometry entities. Mesh geometry entities define the relationship between mesh vertices and the (multiple) surfaces on which they are defined. Underlying topology can be determined on these entities with two functions relating curve and surface vertices to topology.

- **`UGCDB_SurfaceVertex_Inquire_Sheet() :`** finds the owning sheet of a surface vertex.

- **`UGCDB_CurveVertex_Inquire_String() :`** finds the owning string of a curve vertex.

Similarly, geometry can be retrieved via:

- **`UGCDB_SurfaceVertex_Inquire_SurfaceGeometry() :`** finds the surface geometry associated with a surface vertex.
- **`UGCDB_CurveVertex_Inquire_CurveGeometry() :`** finds the curve geometry associated with a curve vertex.

Two other functions are provided to identify the relationship between mesh and mesh geometry (vertex) entities.

- **`UGCDB_SurfaceVertex_Inquire_Vertex() :`** finds the vertex associated with a surface vertex, if one exists.
- **`UGCDB_CurveVertex_Inquire_Vertex() :`** finds the vertex associated with a curve vertex, if one exists.

The final group of inquiry functions includes those used to identify mesh geometry, topology and mesh entities belonging to other mesh entities. Mesh geometry belonging to mesh entities is determined with the following functions:

- **`UGCDB_Vertex_Inquire_SurfaceVertices_In_Sheet() :`** finds all surface vertices from the specified sheet that are attached to the specified vertex.
- **`UGCDB_Vertex_Inquire_AnySurfaceVertex_In_Sheet() :`** finds any surface vertex from the specified sheet that is attached to the specified vertex.
- **`UGCDB_Vertex_Inquire_CurveVertices_In_String() :`** finds all curve vertices from the specified string that are attached to the specified vertex.
- **`UGCDB_Vertex_Inquire_AnyCurveVertex_In_String() :`** finds any curve vertex from the specified string that is attached to the specified vertex.

Topological entities associated with a given mesh entity can be retrieved with the four functions below.

- **`UGCDB_Cell_Inquire_Volume() :`** finds the owning volume of a cell.
- **`UGCDB_Face_Inquire_Sheet() :`** finds the owning sheet of a face.
- **`UGCDB_Edge_Inquire_String() :`** finds the owning string of an edge.
- **`UGCDB_Vertex_Inquire_Model() :`** finds the owning model of a vertex.

Finally, the relationships between higher-order mesh elements (edges, faces and cells) and their low-order elements (vertices) are defined by:

- **`UGCDB_Cell_Inquire_Vertices()` :** finds the ordered list of vertices that define a cell.
- **`UGCDB_Face_Inquire_SurfaceVertices()` :** finds the ordered list of surface vertices that define a face.
- **`UGCDB_Edge_Inquire_CurveVertices()` :** finds the ordered list of curve vertices that define an edge.
- **`UGCDB_Vertex_Inquire_Point()` :** finds the model space coordinates of a vertex.
- **`UGCDB_Cell_Inquire_ElemType()` :** finds the type of a cell.
- **`UGCDB_Face_Inquire_ElemType()` :** finds the type of a face.
- **`UGCDB_Edge_Inquire_ElemType()` :** finds the type of an edge.
- **`UGCDB_Vertex_Inquire_TopologicalType()` :** finds the topological type of a vertex.

Note that in combination with UGCDB_CurveVertex_Inquire_Vertex() and UGCDB_SurfaceVertex_Inquire_Vertex(), this set of functions provides direct access to all pertinent mesh data.

For completeness sake, a couple of generic querying functions are also provided.

- **`UGCDB_Entity_Inquire_Class()` :** finds the class of an entity.
- **`UGCDB_Session_Inquire_VendorDescription()` :** finds a description of the vendor of the current library.

7.4.1.2 Parameter Data Functions

Included in the database library API are four functions for the establishment and querying of generic data. Each of these functions uses a **`UGCPD_`** prefix, identifying it from the standard database functions that access/store data for a specific purpose. The parameter data allows any addressable entity to have a key-value pair associated with it, where the key is a character string and the value is either a Boolean, integer, real, enum or character string. The key is used to identify the parameter to be assigned, and the value is what it is assigned.

- **`UGCPD_Entity_Assign_ValueToKey()` :** assigns a value corresponding to a key for a parameter associated with the specified entity.
- **`UGCPD_Entity_Inquire_Keys()` :** finds all the keys currently associated with the specified entity.

- **`UGCPD_Entity_Inquire_ValueFromKey()` :** finds a value corresponding to a key for a parameter associated with the specified entity.
- **`UGCPD_Entity_Remove_Key()`** : removes a key/value pair from the specified entity.

The parameter data functions may be used for any conceivable purpose required by the application or the meshing library, with all parameter data stored within the database library memory.

One important use of parameter data is for the establishment and maintenance of tunable data communicated to the meshing library upon execution of meshing algorithms. An algorithm within a meshing library must advertise all of its tunable parameters via functions described with the meshing library functions (Section 7.4.2). Each tunable parameter is given a character-string name and returns a data structure containing information such as the parameter variable type and its range of valid values. When the meshing algorithm is executed, it will search database memory for parameter keys matching the tunable parameters associated with the algorithm. Where a match is found, the meshing algorithm bypasses its default value, using instead the value assigned to the parameter. It is the responsibility of the application developer to assign character strings to parameter keys that are identical to those required by the specific algorithm.

7.4.1.3 Plugin Library Functions

The ability to access multiple meshing libraries simultaneously requires a mechanism for differentiating between functions of the same name in separate libraries. Though algorithms satisfying this capability are not specified nor recommended within the API, a sample implementation using the GLib library is provided in Section 9.6.4. This sample implements the two functions below:

- **`UGCP_Session_Load_Plugin()`** : loads a plugin instance of a UGC library into a structure allowing access to a specific instance of every UGC function.
- **`UGCP_Session_Unload_Plugin()`** : unloads a plugin instance of a UGC library.

The **`UGCP_Session_Load_Plugin()`** function passes in the name and location of the UGC-compliant meshing library to use. Passed back is a data structure containing pointers to *all* functions offered within that library. With this data structure, applications may access functions from specific libraries by prefacing the call with the data structure variable assigned to that library. A more detailed description of this feature’s usage is offered in Section 7.5.5.

7.4.1.4 Geometry Handling

Most curve and surface meshing algorithms allow the meshes they generate to be constrained to geometry, defined either by spline representations such as NURBs or by discrete data such as triangular surfaces. While the extent of the geometry functions required to construct these meshes is fairly small, there is usually considerable infrastructure required in support of these functions, and they are often part of larger, more comprehensive libraries such as proprietary CAD packages. This fact makes the likelihood of a meshing library with geometry support rather unlikely.

Therefore, the UGC meshing library specification *does not* require geometry functionality as part of the API. Rather, geometry functions are passed into the meshing library via function pointers specified by the application. This permits the application developer to insert the preferred/available geometry engine into the application, leaving him/her to concentrate on meshing algorithms rather than geometry issues.

Application-specific geometry functions are communicated to the meshing library through the **`UGCDB_Sheet_Attach_Surface()`** and **`UGCDB_String_Attach_Curve()`** functions. These functions are also used to assign geometry itself to the surfaces and strings. These functions take sheet and string entities as input, implying that is possible to specify the geometry functions at the individual entity level. Central to these functions are the **`UGCG_Surface`** and **`UGCG_Curve`** structures, each containing pointers to pertinent geometry functions as well as an opaque pointer into the application-specific structures/arrays for geometry storage. It is the application's responsibility to insure that these opaque values assigned to the sheet/string entities can be dereferenced into meaningful identifiers within the supplied geometry functions. By attaching the UGCG_Surface or UGCG_Curve structures to the sheet/string, the user is specifying the geometric entities to which the meshes will be constrained, and also the functions required to process those entities.

For surface meshing, functions for surface evaluation (determine a point's physical location from its parametric coordinates) and projection (snap a point onto a geometric entity) may be specified. Note that there is no assumption made regarding the method of projection or evaluation employed. For curve meshing, both evaluation and projection function pointers may be specified. A third function may be specified that will evaluate a 1-D point on a curve into a 2-D point on the surface adjacent to the input curve.

Geometry is linked to mesh elements (specifically, vertices) via the **`SurfaceVertex`** and **`CurveVertex`** mesh geometry entities, each of which will contain a **`UGCG_SurfaceGeometry_s`** and **`UGCG_CurveGeometry_s`** data structure, respectively. Surface vertices are defined by a surface (generally a more-specific subsurface than the abstract surface assigned to the sheet), a 2-D location in parametric space, partial derivative values in each of the parametric directions, a unit surface normal, and values pertaining to the minimum and maximum curvatures of the point on the surface. Curve vertices are defined by a smaller data set, including a subcurve, a 1-D location in parameter space, a partial derivative in the parametric direction, a unit normal pointing towards the center of curvature, and the curvature at the point.

7.4.2 Meshing Library

The meshing library is used to query the capabilities and parameters of the library, to apply optional meshing functionality, and to execute the meshing algorithms offered in the library. It is an intentionally leaner API than the database library, providing potential library developers with a quick and straightforward route to UGC compliance. Its functions are defined within a single header file, ugc_mesh.h.

7.4.2.1 Querying Functions

A suitable meshing library API will minimally require functions for the generation of curve, surface and/or volume meshes. Support for even these minimal mesh generation functions is complicated by the fact that most methods rely on carefully tuned run-time parameters. Further, more advanced meshing libraries may have the ability to perform peripheral tasks such as adaption, quality measuring, and decimation. Rather than attempt to encapsulate all foreseeable tunable parameters and peripheral mesh operations, the meshing library API is designed in a way that allows for generality and extensibility.

Since the tunable parameters and peripheral functions are not specified by the API, it is the responsibility of the meshing library to publish its entire set of capabilities. While separate documentation of the meshing library is highly encouraged, broadcast of the library's features will be procedural as well

Three session-level functions are provided for basic information about the particular meshing library.

- **`UGCM_Session_Inquire_VendorDescription() :`** finds a description of the vendor of the current library, as well as the vendor of its database library for direct meshing purposes.
- **`UGCM_Session_Inquire_StatusDescription() :`** finds a description of a status value in the current session.
    - **`UGCM_Session_Inquire_Algorithms() :`** finds the algorithms supported by the current session.

The first of these functions provides the application developer with a minimal idea of the library's origin and possibly version number. The second is used to translate the meshing library's error codes into text descriptions. The last of these three functions is used to alert the application of all available algorithms. While no specific guidelines for the algorithm names are prescribed in the API, the algorithms should be given self-descriptive names inasmuch as is possible.

Five more functions are provided to assess the list of tunable parameters available for a given algorithm.

- **`UGCM_Algorithm_Inquire_Parameters() :`** finds the names of the input and output parameters for the given algorithm, possibly limited to a specific entity.
- **`UGCM_Algorithm_Inquire_Properties() :`** finds the properties of a specific algorithm.

- **UGCM_AlgorithmProperties_Destroy() :** destructor for a UGCM_AlgorithmProperties structure.
- **UGCM_Parameter_Inquire_Properties() :** finds the set of allowable values corresponding to a key for an optional parameter on the specified entity.
- **UGCM_ParameterProperties_Destroy() :** destructor for a UGCM_ParameterProperties structure.
- **UGCM_Algorithm_Inquire_Parameters()**returns a character-string list of the parameters that are associated with a given algorithm.

UGCM_Parameter_Inquire_Properties() returns the properties associated with a given parameter. These properties include a list of supported entity classes, the data type of the parameter, its default value, and the range of valid values (or a list of enum values). In order for an application to modify the current value of a given parameter, the functions described previously in Section 7.4.1.2 must be employed, using key values exactly equal to the parameter text string to be modified.

UGCM_Algorithm_Inquire_Properties() is used to find all of the properties associated with an algorithm. A property lists the supported entity classes that can be passed to the algorithm, which geometry functions need to be defined prior to algorithm usage, the extent of surface information required, and the types of elements required for input and output.

7.4.2.2 Option Functions

There are two optionally supported features of the meshing library API that may enabled via function calls.

- **UGCM_Algorithm_Register_AlternatePlugin() :** registers, in a meshing library, an alternate plugin to use for a specific meshing algorithm.
- **UGCM_Session_Use_DirectMeshing() :** tells the current meshing library that the database library uses the same internal representation and can bypass the formal UGCDB_ interface for optimization purposes.

The first of these functions, described in more detail in Section 7.5.5, will return a value of **UGCM_STATUS_UNSUPPORTED** if alternate plugins are not available in the library. Likewise, the second function, described in more detail in Section 0, will return a value of **UGCM_STATUS_UNSUPPORTED** if direct meshing is not available in the library.

7.4.2.3 Execution Functions

A single function is used to execute an algorithm available in the meshing library.

- **UGCM_Entity_Execute_Algorithm() :** executes a meshing algorithm on a entity in the current database.

This function requires an entity and algorithm as input, at which time the algorithm will be executed subject to the algorithm parameter settings assigned by the application.

7.5 Usage Scenarios

The dual library layout and the specialized functions described above allow the UGC API to be used in a variety of different scenarios. The more important of these methods are outlined in this section, in order of increasing complexity. The first three involve integration of a single meshing library into the user's application. These are expected to be the most prevalent library applications, particularly for new users. Two multiple meshing library applications are also covered.

7.5.1 Meshing Library Only

The simplest conceptual means of employing the UGC API is to build an application around an existing meshing library. This approach is illustrated in Figure 2.

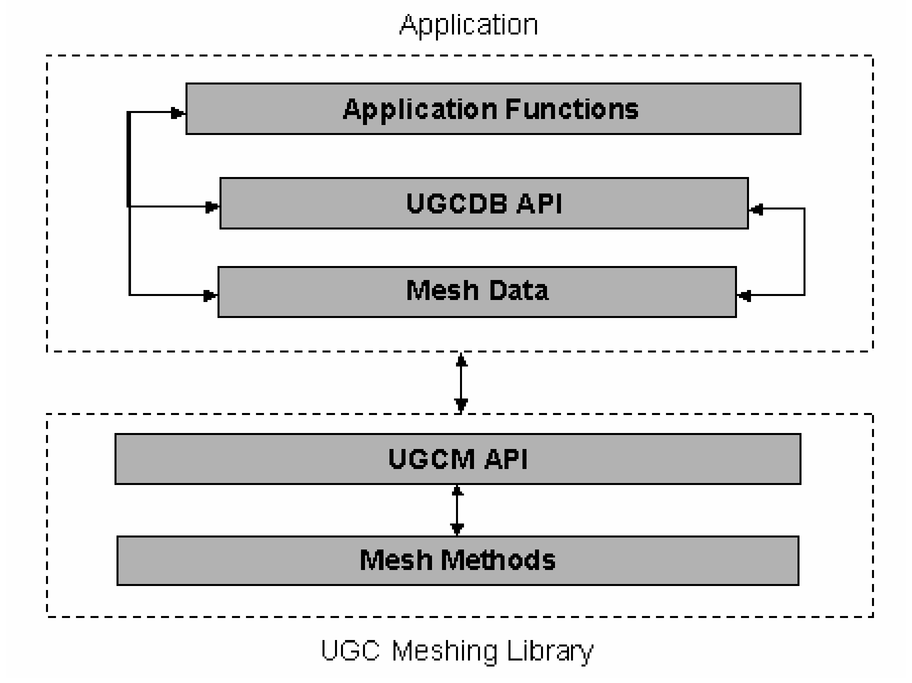


Figure 2. Meshing Library Only

Here, the user develops functions that serve as entry points into the meshing library API. Though these functions must adhere to the database library API protocol, they do not reside within a standalone database library. Instead, they are integrated directly into the application. One advantage of this approach is that the mesh data transfer functions (database library API emulations) directly transfer data to and from the application data structures. This can reduce the number of local copies of the data but may be impractical if the application data structures differ greatly from the UGC model data format. This direct approach requires the greatest amount of work on the part of the developer since he/she must develop code that implements all of the database library functions

7.5.2 Database and Meshing Libraries

A second approach is to implement the data transfer functions (database library API) in a standalone module, separate from the application and the meshing library, as illustrated in Figure 3. The database library serves as a mesh repository that brokers the transfer of mesh data between the application and the meshing library. In this approach, mesh data is accessed directly only by database library functions, with both the application and mesh libraries accessing mesh data indirectly through the database library API. This arrangement will likely prove to be the most popular, since it allows application developers to bypass significant effort by employing existing database and meshing modules. However, this scenario also results in the greatest amount of redundant data, since copies of grid entities may exist in the application, database, and mesh libraries simultaneously.

In order to encourage and stimulate UGC API usage among its target audience, a publicly available database library has been developed for dissemination in source code form (see Chapter 9). The database module approach is well suited to developers retrofitting existing software, and will provide a robust starting point for subsequent customized versions of the UGC database library.

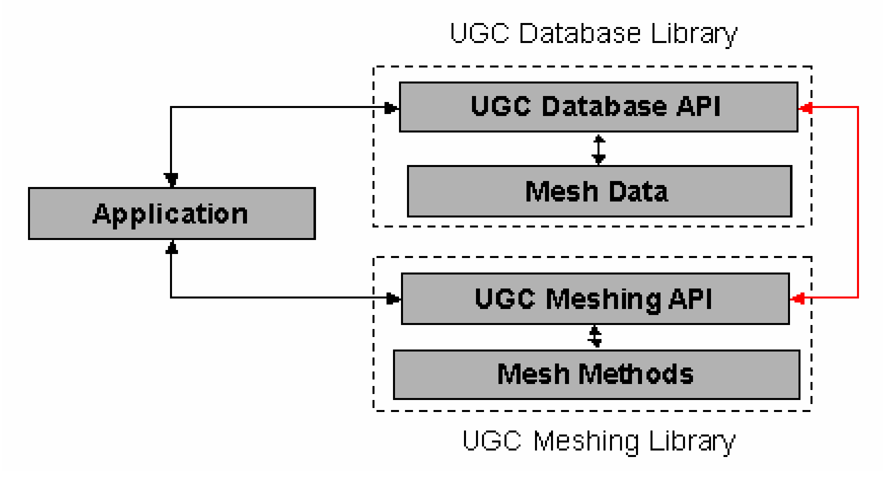


Figure 3. Database and Meshing Libraries

7.5.3 Database and Meshing Libraries with Direct Meshing

To eliminate the data duplication problem inherent in the previous approach, a special function has been incorporated into the meshing library API that provides for direct access of the mesh data stored in database library memory. Proper use of this function, **`UGCM_Session_Use_DirectMeshing(),`** is dependent on the meshing library being fully aware of the internal data format employed by the particular database library to which it is linked. As such, it is not intended for general usage, but rather only for those applications for which duplication of memory is not an option.

The utility of this approach, referred to as Direct Meshing, is best illustrated by example Figure 4. Consider a scenario where a database library holds all mesh data, and a meshing library is used to create a volume mesh, perform a mesh quality analysis and then adapt the mesh. Large dataset transfer is required at least three times, once for the transfer of the surface grid data to the meshing module, once for transfer of the volume data from the meshing library back to the database module, and one additional transfer (adapted mesh) of the volume mesh from the meshing to the database library. Now consider the same library arrangement with direct meshing. Here, the meshing library would work directly on the surface grid datasets residing in the database modules, and would directly populate the database library with the volume mesh, both originally and during the adaptive sweep. In this case, no superfluous transfer of data is required, thereby reducing memory usage and increasing throughput.

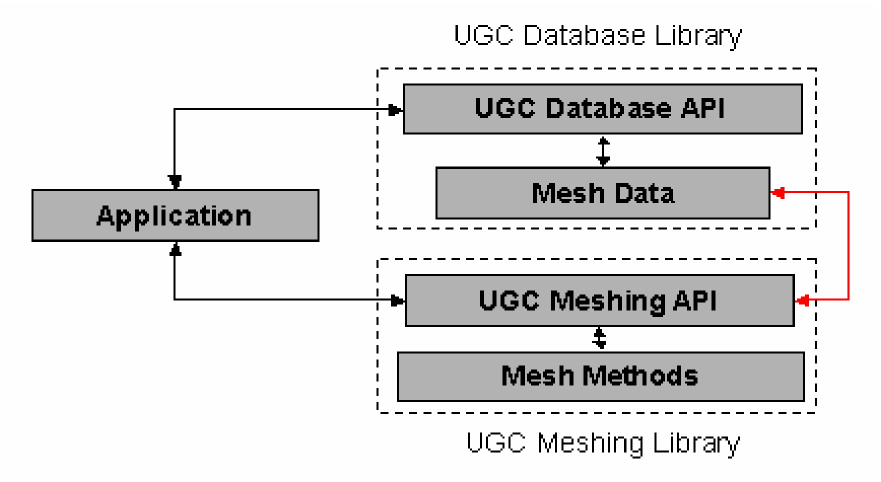


Figure 4. Database and Meshing Library with Direct Meshing

The ability for a database and meshing library pair to communicate on a direct access level is dependent on the meshing library's full knowledge of the particular database library's data storage format. Since the actual format is not part of the API specification (though an example implementation is offered), successful usage of this technique within the meshing module likely requires source code access to the database module to which it is tied.

The API specification mandates that all meshing libraries support the normal transfer mode, whereby all mesh data is transferred via database function calls. This mandate guarantees that meshing libraries will be truly interchangeable. As a specification for direct meshing support, each function must possess special coding that bypasses the formalized API calls, instead working directly on the database library data, ostensibly controlled by conditional statements. Example code using direct meshing is provided in the Users Manual.

Application developers may query a given mesh library via the **`UGCM_Use_Direct_Meshing()`** function to determine whether or not direct meshing support is available. A returned value of **`UGCM_STATUS_UNSUPPORTED`** indicates that direct meshing is not supported. If **`UGCM_STATUS_NOERRORS`** is returned, the developer must then insure that the meshing library employed is compatible with the database library in use. This is done by extracting vendor identification strings from each library and comparing. **`UGCDB_Session_Inquire_Description()`** will return the database library identifier, and **`UGCM_Session_Inquire_Description()`** will return the meshing library identifier and the identifier of the database library it was linked against. If the two database strings appear to be compatible, it may be permissible to use direct meshing. The application developer should also consult the meshing library documentation to determine compatibility with the database library.

7.5.4 Database and Multiple Meshing Libraries

One of the primary benefits of the UGC API is that it provides the application developer with access to a wide assortment of mesh generation tools and capabilities. In many instances, these tools will be obtained eclectically, leaving the application developer with multiple instances of UGC-compliant meshing libraries. Concurrent or serial usage of multiple meshing libraries within a single application is illustrated in Figure 5. The developer's need for multiple library access can arise when one library does not support all of the needed capabilities required by the application, or when the developer wishes to combine capabilities across libraries. For example, Library 1 may be used for curve and surface meshing while Library 2 is used for volume meshing.

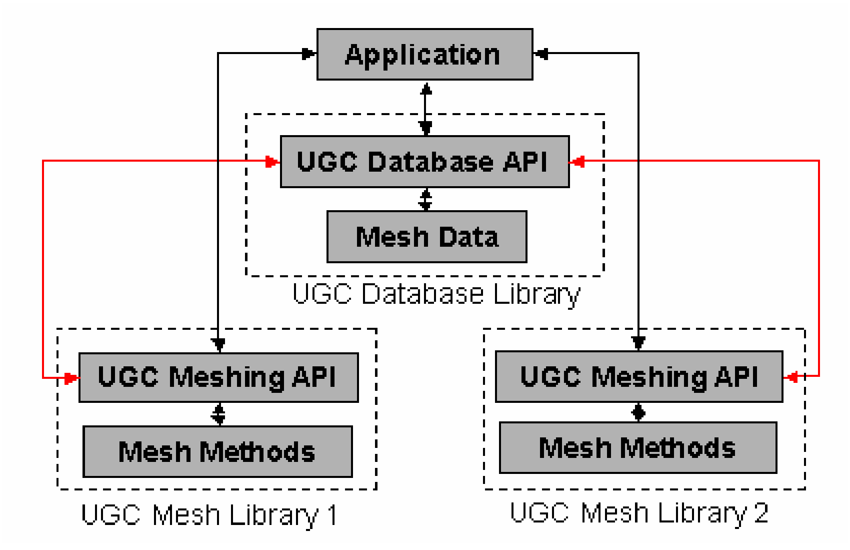


Figure 5. Database and Multiple Meshing Libraries

Such a capability allows a process to be customized from algorithms in multiple libraries. The application developer is tasked with ensuring compatibility between the meshing libraries, however. For instance, in the previous example, if the volume mesh in Library 2 requires triangular faces as input, the application must ensure that the surface methods employed in Library 1 will indeed create triangular elements. This can be determined with the **`UGCM_Algorithm_Properties()`** API function.

Multiple meshing libraries are maintained via the **`UGCP_Session_Load_Plugin()`** and **`UGCP_Session_Unload_Plugin()`** functions. To load a particular dynamic library, the name and location of the shared object (.so or .dll, typically) is input, and returned is a `UGCM_Plugin_s` data structure containing pointers to all meshing library API functions. The application can then call these functions directly. For example, if surface meshing is required from Library1 and volume meshing is required from Library2, calls similar to `MeshLib1->UGCM_Entity_Execute_Algorithm(entity, “surface mesh”)` and `MeshLib2->UGCM_Entity_Execute_Algorithm(entity, “volume mesh”)` would be made. The `UGCP_Session_Unload__Plugin()` is then called when the particular mesh model is no longer required.

A publicly available plugin library has been developed for dissemination in source code form (see Chapter 9). It manages multiple instances of functions using the publicly available GLib library.

7.5.5 Database and Meshing Libraries with Alternate Plugins

In the previous scenario, multiple meshing libraries act independently in the sense that each is unaware of the other, with all code flow control maintained by the application. In general, however, it may be necessary for algorithms in one library to access methods present in another.

In this situation, each library acts as an application with respect to the other library. For instance, the volume mesh generation algorithm in Library 2 may regenerate a portion of the surface mesh during the generation of the volume mesh. Due to limitations or personal preference, the application developer may desire the surface meshing called in Library 1 to be performed by the algorithm in Library 2. The necessary library relationships for this situation are represented schematically in Figure 6.

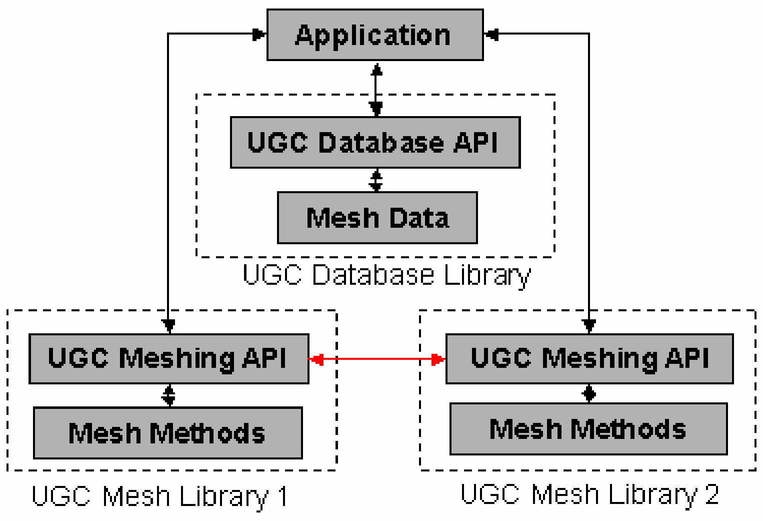


Figure 6. Database and Multiple Meshing Libraries with Alternate Plugins

The `UGCM_Algorithm_Register_AlternatePlugin()` API function is used to specify that alternate functions be used for certain algorithmic applications. Suppose that Library1 is used for volume meshing, but would like to use a surface-meshing tool within Library2 as part of its volume mesh generation process. The application first loads

Library2 as a plugin module via **`UGCP_Session_Load_Plugin(“/usr/lib/lib2.so”,&Lib2)`**. This creates a series of handles to the API functions within Library2. Next the application identifies the algorithm (e.g., **`SurfAlg2`**) from Library2 to be used within Library1 whenever its volume mesh algorithm (e.g., **`VolAlg1`**) is called, and registers it via **`UGCM_Algorithm_Register_AlternatePlugin(“VolAlg1”,”SurfAlg2”,Lib2).`** If Library1 does not support alternate plugins, **`UGCM_STATUS_UNSUPPORTED`** will be returned. Now, when **`UGCM_Entity_Execute_Algorithm(Entity,“VolAlg1”)`** is called within the application, Library1 will in turn call **`Lib2->UGCM_Entity_Execute_Algorithm(Entity,“SurfAlg2”)`** for surface meshing in place of its own algorithms. As in previous scenarios, it is the application’s responsibility to insure that the algorithms assigned via alternate plugins are compatible with the primary algorithms with regard to element types.

7.5.6 Others

All combinations of multiple mesh libraries, direct meshing and alternate plugins are also supported in the API, though they will not be described explicitly. These include:

- Mesh library with direct meshing and alternate plugins
- Multiple mesh libraries with direct meshing
- Multiple mesh libraries with alternate plugins
- Multiple mesh libraries with direct meshing and alternate plugins

8 Demonstration

The feasibility and utility of the API V2 design is demonstrated in this chapter through simple and complex examples. First, several individual API features are demonstrated via pseudo-code intended to illustrate their usage. In a considerably more complex example, an API V2-compliant meshing library was integrated into two separate API V2-compliant applications. Both the meshing library and the applications in this latter example were converted from separate proprietary mesh generation packages.

8.1 Feature Demonstration

During the two-year design and development of UGC API V2, it was sometimes necessary to develop pseudo-code examples of particular features. These examples were helpful for the contract investigators to explain specific feature implementations to the design team. These files were retained and updated regularly to reflect interim design changes, and the collection of these code examples are distributed with the User Manual.

Features demonstrated include geometry-constrained surface meshing, dynamic library maintenance via the plugin functions, and example usage of direct meshing.

8.2 Integration Into Existing Applications

Several demonstrations were performed to investigate the usability and efficiency of the API design when implemented in a legacy mesh generation application and library. These demonstrations included unstructured surface mesh generation on various types of geometry surfaces including: a single parametric surface, multiple parametric surfaces, and a periodic parametric surface. Surface meshes of various sizes were generated to test the efficiency on large meshes. The ability to interchange a mesh library between two separate conforming applications was demonstrated along with the ability to swap libraries during execution using the UGC plug-in facility.

To perform these demonstrations, two existing mesh generation applications and an existing mesh generation library were converted. The purpose of these demonstrations were to test the efficacy of the API specification, document the process for modifying existing code, measure the amount of labor involved in implementing the specification, measure the overhead in using the API, and demonstrate that the API could be used in a production mesh generation tool.

8.2.1 Database Library Development

The largest single coding effort in this demonstration was the development of a database library, due primarily to the sheer number of functions to be implemented, and to the fact that new data structures needed to be devised. Once this library was completed and validated, however, it became apparent that it would be of use to a potentially large number of mesh library developers, particularly those whose needs are within the scope of the API V2 effort. The decision to deliver this sample database library with the standard API V3 distribution was made shortly thereafter. Detail regarding this utility library may be obtained in Section 9.6.2.

8.2.2 Fortran Wrapper Library Development

The MADCAP mesh generator (See Section 8.2.4) is written in the Fortran 90 (F90) programming language while the UGC API is based on the C programming language. The application developer is responsible to make sure his application communicates properly with the API specification. In this instance, the application developer must implement a method for communicating between the Fortran and C languages. One approach is to write a C interface that wraps around the API function call to provide inter-language communication. The wrapper function must handle all of the necessary conversions between languages. This approach was taken to develop a library of Fortran to C wrappers for all UGC database and meshing API functions. A general approach was taken to develop the wrapper library that would work with a wide range of compiler and machine architectures.

The success of this library during this demonstration forged the plan to deliver the library as part of the API V2 distribution. This library, explained in more detail in Section 9.6.3, allows developers to use the UGC library within a Fortran application simply by compiling and linking the library and by calling API functions within the application by their translated form.

8.2.3 Mesh Library Conversion

For this demonstration, the Advancing Front with Local Reconnection (AFLR) library was modified to conform to the UGC specification for mesh libraries. The AFLR library performs two- and three-dimensional unstructured mesh generation and was written by David Marcum at Mississippi State University [24]. Only the surface mesh algorithm was implemented for this demonstration. This effort consisted of implementing the 10 UGC meshing API functions, and developing functions to retrieve the boundary loop(s) from the database and load the resulting surface mesh back into the database. All of the necessary modifications were isolated to an interface on top of the ALFR library and are contained in the files **`AFLR_UGCInterface.c`** and **`AFLR_UGCLib.c`** provided with the distribution.

A UGC compliant library is required to implement all 10 of the UGC Meshing functions. Most of these functions return information about the library in the form of text strings. Information about library capabilities, input expectations, and resulting output data can all be retrieved through these interfaces. For this demonstration, the UGC compliant meshing library capability was limited to the generation of triangular surface meshes on parametric geometry. The limited scope reduced the development effort by only requiring information about one algorithm to be returned in the UGC meshing functions. The AFLR library implementation of the UGC meshing functions can be found in the file **`AFLR_UGCLib.c`** file in the distribution.

Most of the meshing library implementation work is contained in the function **`UGCM_Execute_Algorithm`** (see the file **AFLR_UGCLib.c**). This function checks the algorithm that is requested against available algorithms in the library. It then retrieves the appropriate data from the database for transfer to the AFLR library. For the ALFR surface meshing algorithm, the **UGCM_Execute_Algorithm** function checks to see that the input sheet is available in the database. It also ensures that the input sheet contains one or more loops (closed collection of strings) and a geometry surface. If the appropriate data is available, then the boundary loops are retrieved from the database, reformatted to the AFLR library data format and loaded into the AFLR data structures. A separate function internal to the AFLR interface library was written to retrieve the boundary loop data from the database and reformat it to the AFLR library format. This function is called **`Get_Loops_From_DB`** and can be found in the file **`AFLR_UGCInterface.c`**. Once the boundary loops have been retrieved, the information is passed onto the AFLR surface-meshing library.

After the AFLR library completes the surface mesh generation, the resulting surface mesh is converted from the AFLR data structures to a format compatible with the UGC database. The surface mesh is then transferred into the database and the mesh memory is freed from the AFLR data structures. The surface mesh is translated from AFLR to the database in the function **`Put_Surface_In_DB`** that can be found in the file **`AFLR_UGCInterface.c`**.

In addition to the implementation of the UGCM interfaces, the AFLR library was also modified to conform to the UGC specification for access to geometry. All references to the geometry surface inside the library were modified to use the geometry callback functions. Two different geometry functions are called by AFLR, one to project to a surface (physical space to parametric space), and another to evaluate a location on a surface (parametric space to physical space). These were modified to use the **`GeometrySurface->Surface_Project`** and **`GeometrySurface->Surface_Evaluate`** callback functions.

8.2.4 Application Conversion and Demonstration

For this demonstration, two mesh generation applications were modified to conform to the UGC API V2 specifications. Example source code passages for the implementation of API V2 into these applications are provided in the Users Manual.

The first application, Modular Aerodynamic Computational Analysis Process (MADCAP), is a production tool for structured/unstructured multi-block and overlapping mesh generation [25]. MADCAP was developed by the Boeing Military Aircraft and Missile Systems organization. The second application, Advanced Pre- and Post-processing Toolkit (APPT), is also a production tool developed for structured/unstructured multi-block mesh generation [26]. APPT was developed by the Boeing Space Systems organization. MADCAP and APPT share many of the same underlying meshing libraries. Integration of the UGC API into these two tools enhances their ability to share future meshing libraries with less development effort.

8.2.4.1 MADCAP

Most of the modifications to make MADCAP a UGC compliant application are contained within the **`UGC2D_Interface`** subroutine, found in the **`UGC2D_Interface.f90`** file. This subroutine performs the following steps:

1. Query library to ensure compatibility
2. Initialize the mesh model in the database.
3. Translate input data to UGC format and load into database.
4. Generate mesh
5. Extract data from database and translate to MADCAP format.

The UGC Fortran wrapper and UGC Fortran 90 libraries were used in the implementation. It is difficult to estimate the effort that was involved in API implementation. The MADCAP implementation was used as a test bed for the API as it was being developed, and hence underwent several revisions during the course of the study. In addition, the database library and Fortran wrapper libraries were being tested/debugged simultaneously with the MADCAP UGC compliant development. Now that the database and Fortran libraries are in hand, a fresh implementation in MADCAP would probably require a few days effort. This implementation was relatively limited (surface meshing only with a known library). It is expected that a general implementation would require slightly more time to implement.

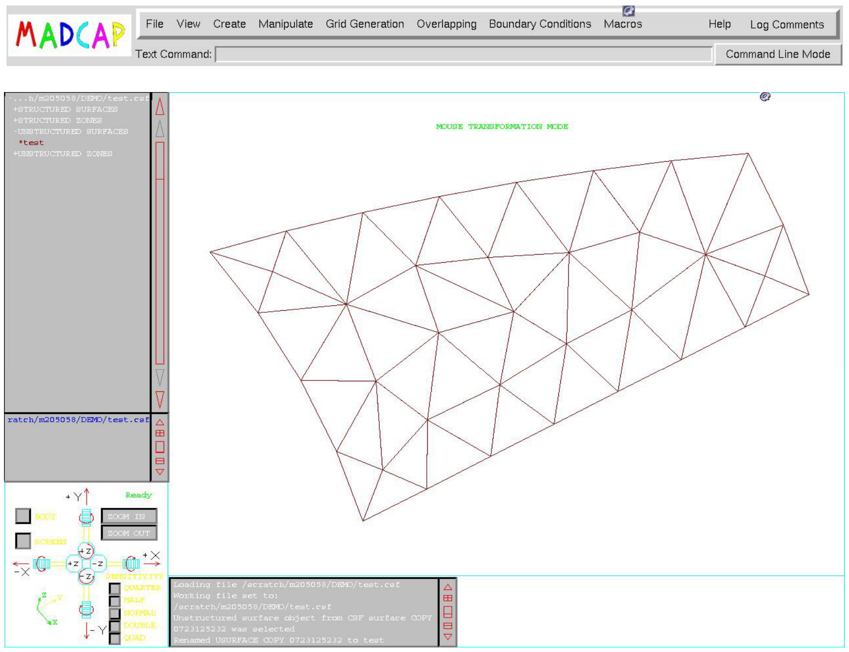
One of the biggest challenges encountered in the implementation was converting the mesh edge and face elements back to a format usable by MADCAP. The face and edge elements returned by the database reference surface and curve vertices through their UGCDB_ID handle. The MADCAP data format references face vertices through their index in the vertex list. Efficient conversion from the database handle to the vertex index required development of hash table. A Fortran 90 module to develop and maintain the hash table was written to perform this task. The source code for the hash table software is in the file **`HashTable.f90`**.

Several test cases were run with the UGC API compliant version of MADCAP. These cases were used to test the API under various scenarios including generation of a single sheet and generation of a mesh model with multiple sheets. The resulting surface meshes for these cases are shown in Figure 7 and Figure 8. The overhead of the API and database were measured by computing a dense mesh on a single surface. The resulting surface mesh contained 19,850 faces and is shown in Figure 9. An even finer mesh with 125,184 faces was also computed. The computation times and memory usage was compared with the non-compliant version of MADCAP and is shown in Table 10.

Test Case	# Faces	MADCAP Version	Memory Usage	Computation Time (sec)
Single Sheet	51	Original	NA	NA
		UGC Compliant	NA	NA
Multi-Sheet		Original		
		UGC Compliant		
Single Sheet Fine 1	19,850	Original		1.78
		UGC Compliant		1.51
Single Sheet Fine 2	125,184	Original		6.82
		UGC Compliant		7.02

Table 10. Comparison of Original and UGC Compliant Application (MADCAP)

An indication of the overhead associated with the UGC API implementation in MADCAP can be obtained from the data in Table 2. There is a slight computational overhead associated with the use of the API. The memory overhead is more significant and varies with the grid size.

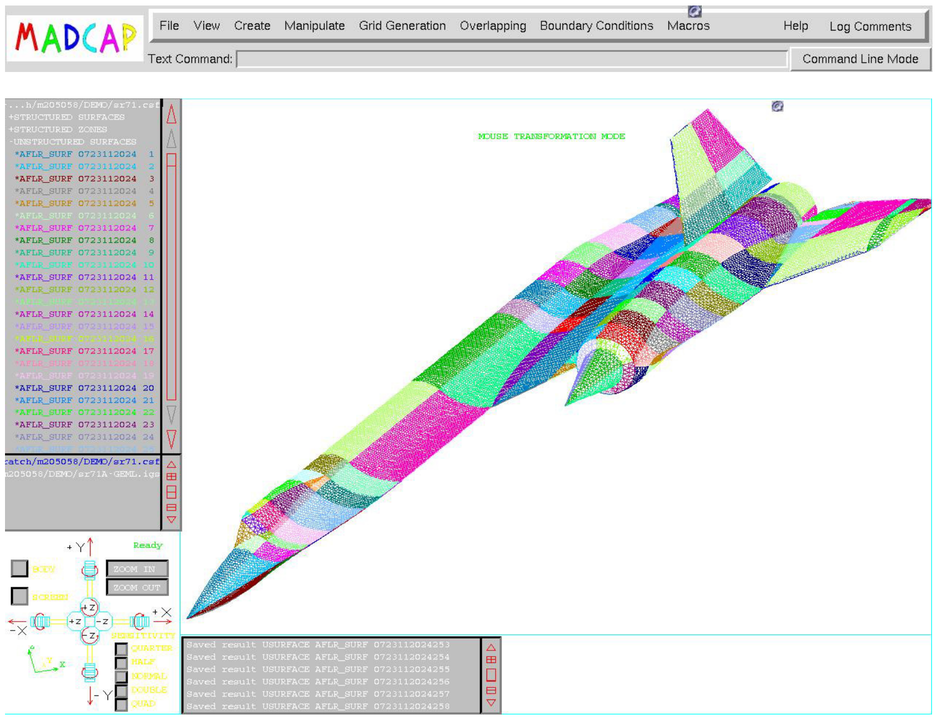


```
-...h/m205058/DEMO/test.csf
 +STRUCTURED SURFACES
 +STRUCTURED ZONES
 -UNSTRUCTURED SURFACES
  *test
 +UNSTRUCTURED ZONES
```

```
ratch/m205058/DEMO/test.csf
```

```
Loading file /scratch/m205058/DEMO/test.csf
Working file set to:
/scratch/m205058/DEMO/test.csf
Unstructured surface object from CSF surface COPY
0723125232 was selected
Renamed USURFACE COPY 0723125232 to test
```

Figure 7. Single Sheet Mesh Comparison in MADCAP

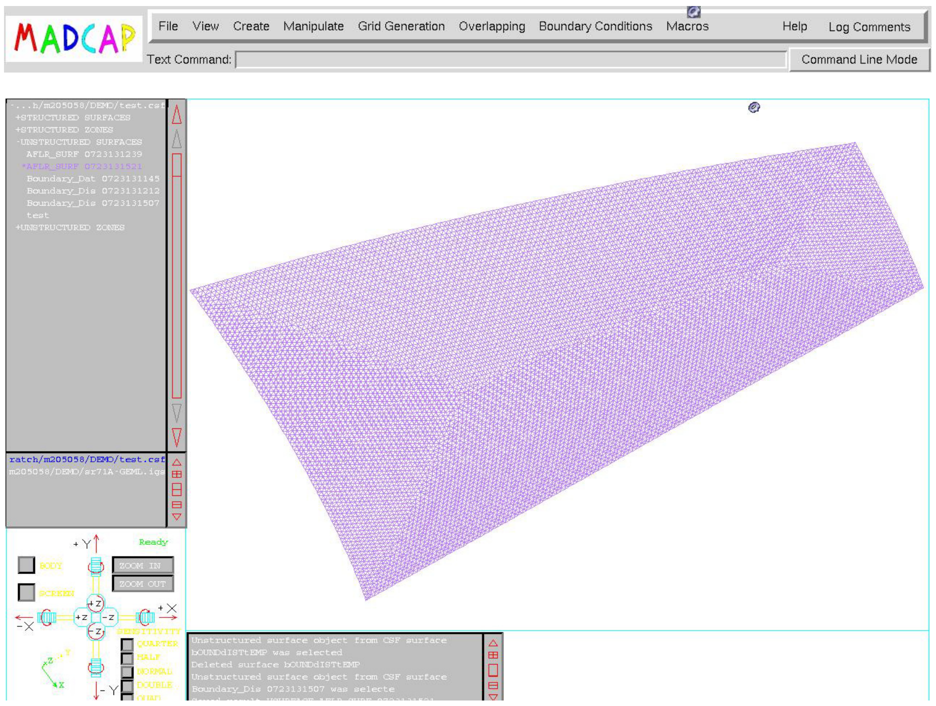


```
-...h/m205058/DEMO/sr71.csf
 +STRUCTURED SURFACES
 +STRUCTURED ZONES
 -UNSTRUCTURED SURFACES
  *AFLR_SURF 0723112024  1
  *AFLR_SURF 0723112024  2
  *AFLR_SURF 0723112024  3
  *AFLR_SURF 0723112024  4
  *AFLR_SURF 0723112024  5
  *AFLR_SURF 0723112024  6
  *AFLR_SURF 0723112024  7
  *AFLR_SURF 0723112024  8
  *AFLR_SURF 0723112024  9
  *AFLR_SURF 0723112024 10
  *AFLR_SURF 0723112024 11
  *AFLR_SURF 0723112024 12
  *AFLR_SURF 0723112024 13
  *AFLR_SURF 0723112024 14
  *AFLR_SURF 0723112024 15
  *AFLR_SURF 0723112024 16
  *AFLR_SURF 0723112024 17
  *AFLR_SURF 0723112024 18
  *AFLR_SURF 0723112024 19
  *AFLR_SURF 0723112024 20
  *AFLR_SURF 0723112024 21
  *AFLR_SURF 0723112024 22
  *AFLR_SURF 0723112024 23
  *AFLR_SURF 0723112024 24
```

```
ratch/m205058/DEMO/sr71.csf
m205058/DEMO/sr71A-GEML.igs
```

```
Saved result USURFACE AFLR_SURF 0723112024253
Saved result USURFACE AFLR_SURF 0723112024254
Saved result USURFACE AFLR_SURF 0723112024255
Saved result USURFACE AFLR_SURF 0723112024256
Saved result USURFACE AFLR_SURF 0723112024257
Saved result USURFACE AFLR_SURF 0723112024258
```

Figure 8. Multi-Sheet Demonstration in MADCAP



```
-...h/m205058/DEMO/test.csf
 +STRUCTURED SURFACES
 +STRUCTURED ZONES
 -UNSTRUCTURED SURFACES
   AFLR_SURF 0723131239
  *AFLR_SURF 0723131521
   Boundary_Dat 0723131145
   Boundary_Dis 0723131212
   Boundary_Dis 0723131507
   test
 +UNSTRUCTURED ZONES
```

```
ratch/m205058/DEMO/test.csf
m205058/DEMO/sr71A-GEML.igs
```

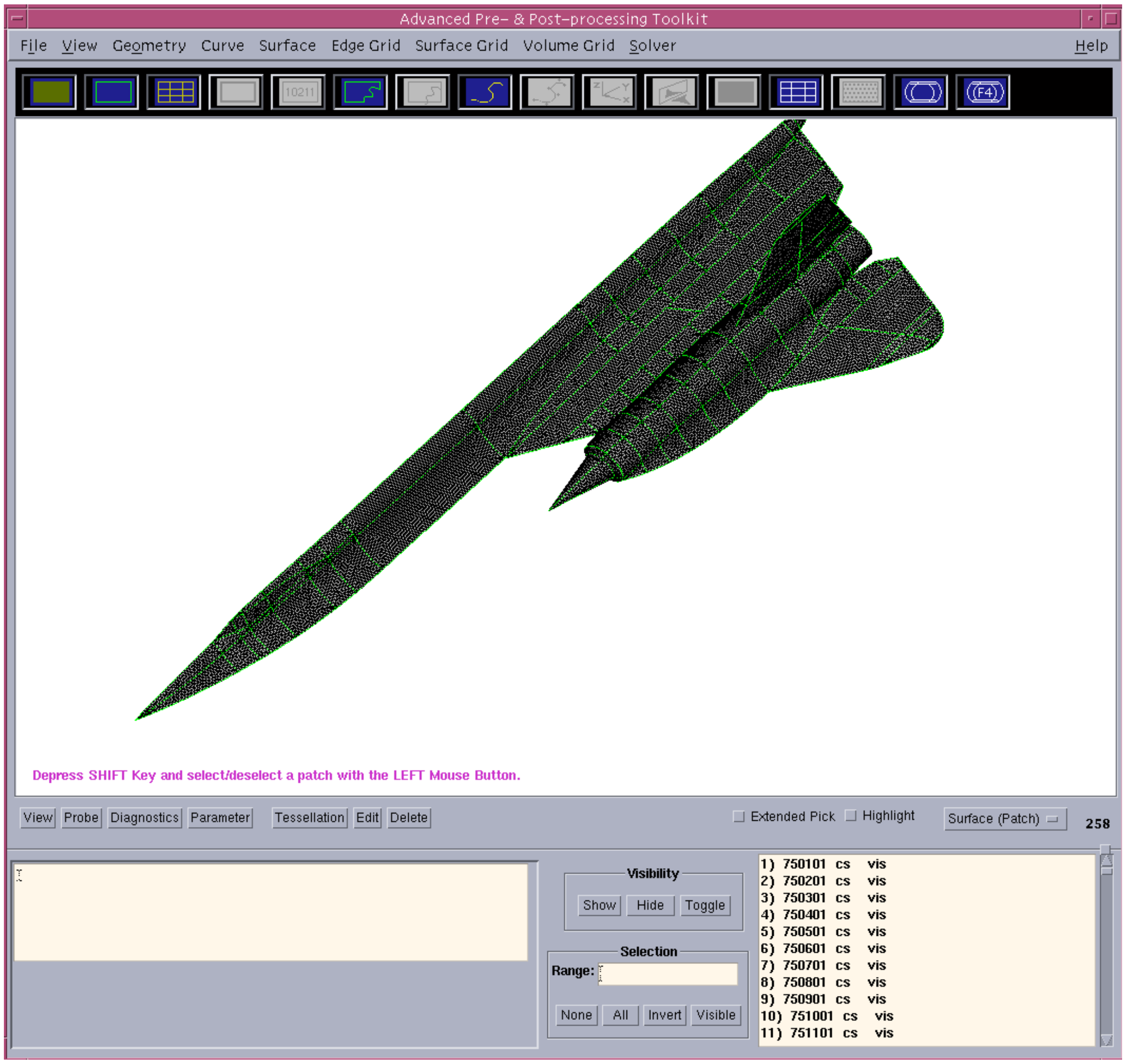
```
Unstructured surface object from CSF surface
bOUNDdISTtEMP was selected
Deleted surface bOUNDdISTtEMP
Unstructured surface object from CSF surface
Boundary_Dis 0723131507 was selecte
```

Figure 9. Dense Single Sheet Demonstration in MADCAP

8.2.4.2 APPT

To demonstrate the ability to share a UGC compliant library between two different meshing applications, the UGC compliant AFLR library was linked with the APPT meshing application. The API implementation paralleled that for MADCAP. In particular, the C++ function **`UGC_aflr2d_interface_CPP`** initialized the mesh model in the database, translated input data to UGC format, loaded the data into the database and generated the surface mesh. Once the mesh was generated, **`UGC_aflr2d_interface_CPP`** called the function **`unload_ugcdbmodel`** to extract the mesh from the database and convert it to a format useable by APPT. A working version of this interface was generated, tested and documented in less than 120 man hours. Of the 120 hours, roughly 16 hours were spent studying the documentation and mapping out a strategy. Eighty hours were spent programming and debugging. The remainder was spent testing and documenting. Programming time encompassed development of **`UGC_aflr2d_interface_CPP`**, **`unload_ugcdbmodel`**, peripheral hash table functions (preliminary versions) as well as modification to the calling functions within APPT. The geometry evaluators and projectors used in APPT are C versions of the analogous geometry evaluators and projectors used in MADCAP. Writing wrappers around the APPT C evaluators and projectors was straightforward. The MADCAP FORTRAN implementation was available as a starting point, and expert consultation from Todd Michal was provided on an as-needed basis (roughly 3-4 hrs). The APPT implementation is basic and does not include the plug-in facility or parameter passing utilities although these improvements should be straightforward.

A variety of sample surface meshes were generated using the APPT UGC API to the AFLR library. These demonstrations are illustrated in Figure 10 through Figure 13.


Advanced Pre- & Post-processing Toolkit
File View Geometry Curve Surface Edge Grid Surface Grid Volume Grid Solver Help
View Probe Diagnostics Parameter Tessellation Edit Delete
Extended Pick Highlight Surface (Patch) 258
Visibility Show Hide Toggle
Selection Range: None All Invert Visible

Depress SHIFT Key and select/deselect a patch with the LEFT Mouse Button.

```
1) 750101 cs vis
2) 750201 cs vis
3) 750301 cs vis
4) 750401 cs vis
5) 750501 cs vis
6) 750601 cs vis
7) 750701 cs vis
8) 750801 cs vis
9) 750901 cs vis
10) 751001 cs vis
11) 751101 cs vis
```

Figure 10. Multi-Sheet Dmonstration in APPT

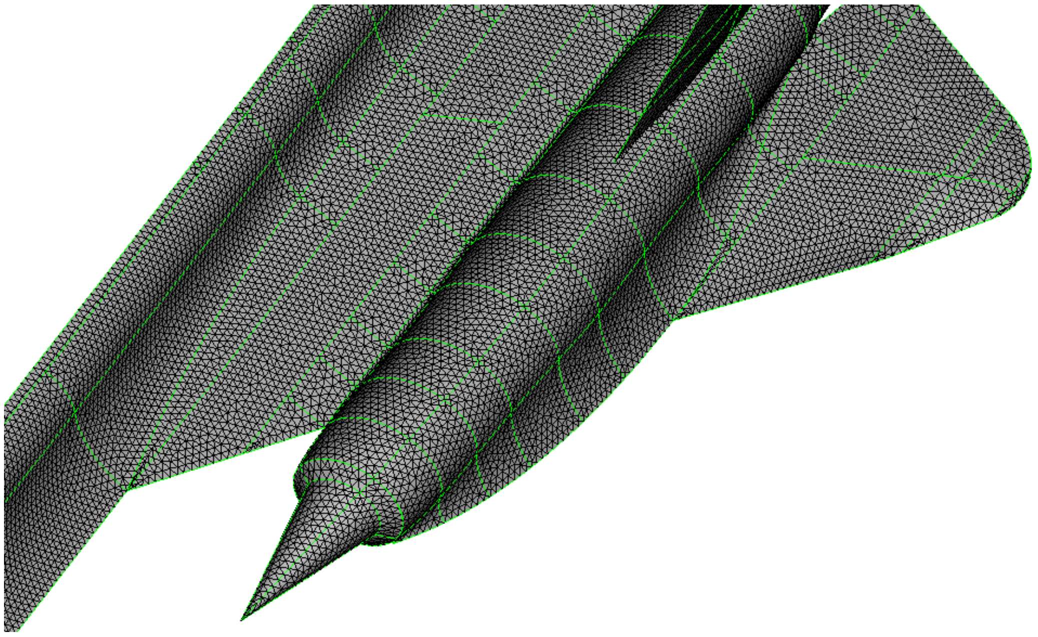


Figure 11. Close-up of Multi-Sheet Demonstration in APPT

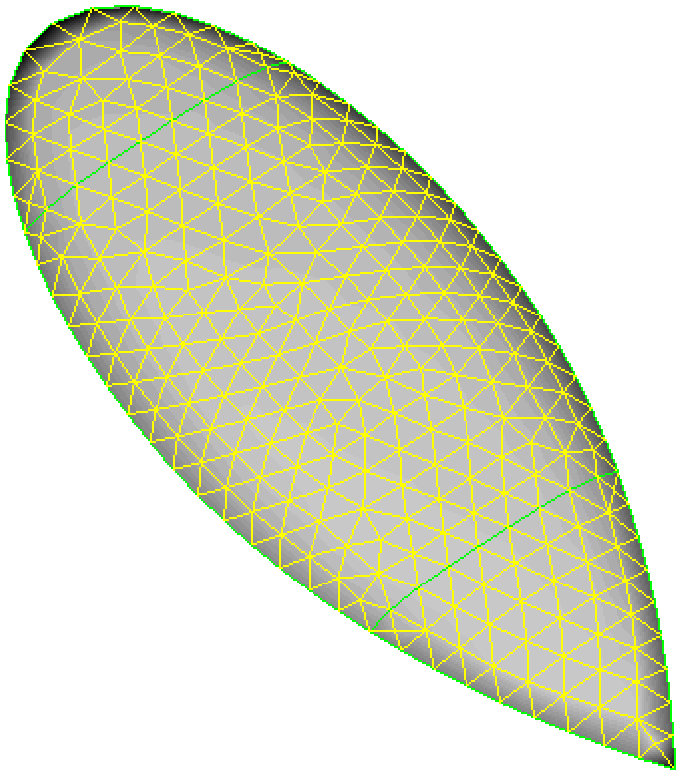


Figure 12. Singular Surface Demonstration in APPT

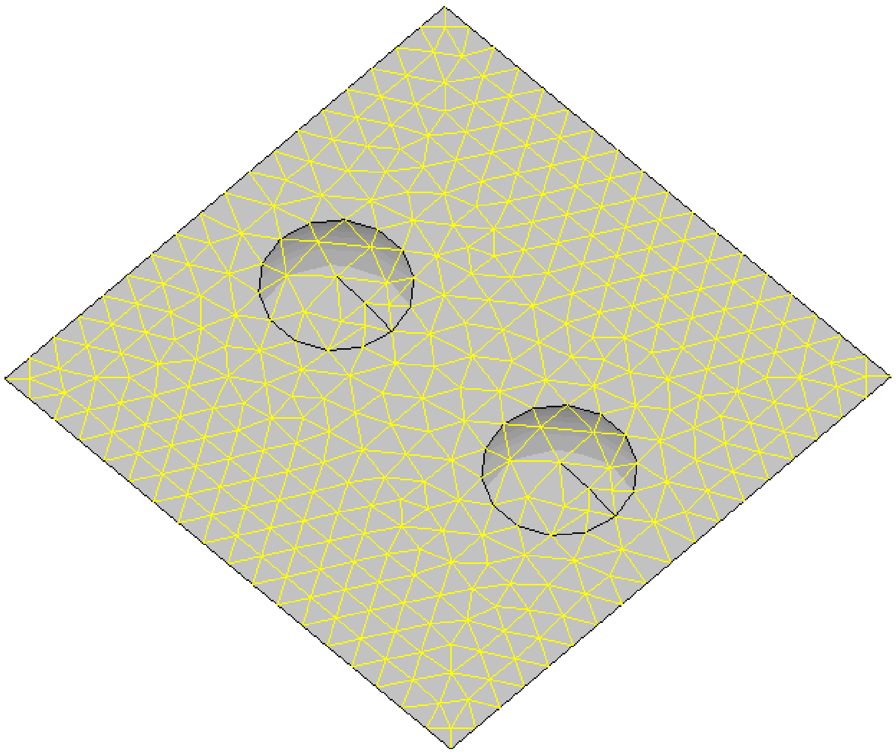


Figure 13. Periodic Surface Demonstration in APPT

The UGC plug-in facility was demonstrated by making two copies of the AFLR UGC compliant library. A simple write statement was added to each library to verify which library was running. The MADCAP application was rebuilt including the GLib libraries and UGC plug-in functions. The application was modified to load library 1 as the plug-in and a surface mesh was generated. Library 2 was then registered as the plug-in and a second surface mesh was generated. The corresponding write messages were used to verify that library 1 and library 2 were dynamically swapped during program execution.

9 Deliverables

For the sake of clarity, each of the subject contract deliverables is discussed in this chapter, with explanations of how and when these required items were satisfied. Additional non-required software items packaged with the deliverables are discussed as well. It is expected that these additional software utilities will immediately increase the viability of API V2 as a standard.

9.1 Data Item A001 - Final Report

The Statement of Work proposed for this effort, included by reference in the executed contract stipulates that two particular items be included in this Final Report.

First, a review of other standards efforts in the area of geometry and meshing relative to the scope of API V1 and relative to accepted computer science practices was to be carried out and included in this final report. These reviews are documented in Chapters 3 and 4.

Secondly, demonstration and documentation of API V2 integrated into an existing application and meshing library was to be included in the Final Report. Two separate demonstrations of this type were conducted and are documented in Section 8.2.4.

9.2 Data Item A002 – Contractor’s Billing Voucher

As required in Data Item A002 of this contract, Contract Billing Vouchers were submitted on a monthly basis to the following agencies: ADMN/ACO, AFRL/CO, AFRL/VAAC, and AFRL/VAF.

9.3 Data Item A003 – Funds and Man-Hour Expenditure Report

As required in Data Item A003 of this contract, Funds and Man-Hour Expenditure Reports were submitted on a monthly basis to the following agencies: ADMN/ACO, AFRL/CO, AFRL/VAAC, and AFRL/VAF.

9.4 Data Item A004 - Presentation Material

The subject contract Initial Briefing was held on 14 July 03 at Wright-Patterson AFB. In attendance from the development team were John Steinbrenner (Pointwise), Todd Michal (Boeing) and Pat Yagle (Lockheed-Martin). Lt. Nopadol Tarmallpark and Matt Grismer represented the Air Force. A copy of the presentation in MicroSoft Powerpoint format was delivered to Lt. Tarmallpark at this time.

9.5 Data Item A005 - Software User Manual

The primary product of this contracted effort is the definition of an API standard for unstructured mesh generation. This API is delivered via a Software Users Manual that documents the functionality of the API. This document is formatted in html, allowing it to be accessed on users’ web browsers. This format also permits copious usage of hyperlinks, allowing users to navigate the document more easily than with a standard printed-page form. Much of the manual is generated procedurally via the Doxygen software [27], a documentation system for many programming languages, including C. Doxygen is freeware that is distributed via the GNU General Public License. As per contract requirements, the User Manual is also delivered in Microsoft Word format.

The contract Statement of Work specifically requires that the User Manual contain a thorough description of the API along with source code examples of simple meshing tasks. Each of these requirements is satisfied in this document.

9.6 Source Code Utilities

Data Items A001 through A005 are the only strict deliverable items in this contract, and each will be satisfied in full at contract completion. During the course of the API design, however, it became apparent that a number of source code utilities could also be provided as part of the API V2 distribution. The inclusion of these utilities in source code form reduces the effort required to construct a UGC-compliant application or library, which in turn increases the likelihood of the API being adopted by potential application and library developers.

These C and Fortran utilities described below are used to perform necessary but relatively generic tasks. Users are encouraged to use any of all of these utilities directly in their implementation of a compliant application or library. The header files, which contain an exact specification of API V2, should not be modified in order to preserve integrity with the V2 specification. The library utilities, on the other hand, represent sample implementations of functional sets. They may be edited at will in order to match the requirements of the particular application. The UGC specification does not require the use of any of these utilities except for the header files, however. The code that is provided has been tested on various applications/hardware platforms, but is not guaranteed to work for a specific application.

9.6.1 Header Files

The exact API V2 specification described in this document and in the User Manual is provided in header file form in the **`dist/include`** directory. This directory consists of five files that should be included directly into C applications and database and meshing library implementations. These header files define functional prototypes, typedefs, data structures, enums and macro variables.

9.6.2 Sample Database Library

The bulk of the work in implementing the UGC API into a new or existing application is in the development of the database library functions. These functions transfer mesh data between the application and library. A complete implementation of the database functions is provided with the distribution in the form of a standalone library. This implementation can be found in the **`dist/database`** directory. The utility library is independent of any specific application or mesh library, which should permit reuse by future developers. By reducing the number of functions that need to be implemented, the database library should significantly decrease the effort involved in implementing the UGC API. The database library was written in the C programming language, and is directly callable from applications written in C or C++. The library data structures mirror the UGC mesh model format (see Figure 1) and supports the simultaneous access of multiple mesh models. The database is fully compliant with the UGC specification and supports all UGC mesh entities and element types. Mesh data can be created, retrieved and freed by the application or library through the UGCDB functions.

A limited amount of checking is built into the database functions to provide a robust and efficient implementation. For instance, the library will disallow the destruction of a mesh entity that is referenced by another mesh entity. All entity handles that are passed into the database are checked to ensure they are appropriate for the operation that is requested. For instance, a request to retrieve face data from a string entity will be trapped and result in an error. If an error is encountered in any of the database functions a standard UGC status code indicating the nature of the error will be returned.

9.6.3 Sample Fortran/C Library

The UGC specification defines the API in the C programming language. Use of the API with another programming language requires that the developer handle the proper communication between languages. One approach for providing this inter-language communication is to write a C interface that wraps around the API function call. The wrapper function must handle all of the necessary conversions between languages. This approach was taken to develop a library of Fortran to C wrappers for all UGC database and meshing library API functions. A general approach was taken to develop the wrapper library that would work with a wide range of compilers and machine architectures. The resulting utility library is provided with the API V2 distribution and can be found in the **`dist/fortran`** directory.

The Fortran wrapper library uses a set of C preprocessor directives to create a common method at the source level to allow for calls between languages. These directives are tailored for each supported compiler to accommodate its particular nuances. While the basic idea of allowing FORTRAN and C to call each other seems straightforward, there are a considerable number of details that must be addressed. These go beyond the appropriate capitalization and addition of underscores to the routine name to include proper treatment of passed variable types, including strings and functions. All of the preprocessor directives have been coded into a single include file called **`bind_f_and_c.h.`** This include file also specifies a correspondence between C and FORTRAN variable types for each supported C compiler. Within the wrapper, several additional transformations are made. The wrapper interface ensures that arguments from FORTRAN are passed by reference, even if values are used in the API function. 2-D arrays are converted by creating a 1-D array of pointers into the 2-D array to simulate the functionality of a C 2-D array. Character strings are converted to properly handle each language’s treatment of the string termination. Subroutine and function names are automatically modified to account for any name mangling performed by the compiler.

Supporting new compilers is a fairly simple task of defining about 20 preprocessor directives. Most of these do not change between compilers, but were used because of special cases. Presently, 17 compilers have been defined using this approach, including UNIX workstations, as well as Linux and Windows NT. Specifically, it has been tested using IBM AIX, Convex OS, Cray compilers, a variety of Hewlett-Packard operating systems, SGI IRIX, Ultrix, VMS, Hitachi, Kendall Square, NEC machines, CDC, Fujitsu, Windows NT, and Linux operating systems.

The source code for the Fortran wrappers is provided in the file **`ugc_fort77.c`**. The interfaces are compatible with the Fortran 77 (F77) programming language with the exception of subroutine name lengths. The UGC API naming convention is followed with an extra extension of _F added to the end of the function name. For instance, the Fortran wrapper to the **`UGCDB_Session_Create_Model()`** function is named **`UGCDB_Session_Create_Model_F().`** The wrapper interfaces are F77 subroutines. These do not return a status but instead return the status through a variable in the argument list. The first argument in all Fortran interfaces is the status variable. The rest of the wrapper argument lists follow the UGC API definition to the extent possible. Exceptions to this rule are made for arguments of a data type that are not compatible with Fortran. C structures are broken into multiple arguments, one for each structure member. Arguments that are an enumerated type are treated as integers in the Fortran interface. Fortran parameters for each of the valid enumerations are provided in the Fortran include file **`ugc_fort77.inc`**. The integer variables can be assigned to the parameter corresponding to a specific enumeration and passed to the Fortran interface. Inside the wrapper, the integer will be converted to the appropriate enumeration and passed on to the C function. An example of the Fortran equivalent call for the API function **`UGCDB_SurfaceVertex_Assign_SurfaceGeometry()`** is provided in Table 11.

C Interface	**`UGCDB_SurfaceVertex_Assign_SurfaceGeometry`**
	 `(UGCDB_ID SurfaceVertex, UGCG_SurfaceGeometry *SurfaceGeometry)`

Fortran Interface	**`UGCDB_SurfaceVertex_Assign_SurfaceGeometry_F`**
	 `(status, SurfaceVertex,  subsurface,  uv, dSdU, dSdV, N, MinCurvature, MaxCurvature, Principal)`

Table 11. Fortran and C Interfaces to UGCDB_SurfaceVertex_Assign_SurfaceGeometry

In the example presented in Table 11, the Fortran interface differs from the C interfaces in the following ways:

1. The function name includes the **`_F`** extension
2. A new argument has been added to hold the status code
3. The **`SurfaceGeometry`** structure has been broken into its members subsurface, **`uv, dSdU, dSdV, N, MinCurvature, MaxCurvature`**, and **`Principal`**.

Fortran77 does not have a pointer variable type equivalent to the C pointer. Therefore, pointers cannot be passed through the argument list to be allocated by the function. For the F77 wrappers, the data blocks must be pre dimensioned to the appropriate size and the dimensioned size passed into the function. The wrapper will automatically handle the transfer of data from the C data blocks to the pre-allocated Fortran array.

Many of the limitations in the F77 wrappers can be avoided if the application is written in F90. Pointer variables and data structures are both supported under F90, although they are not interchangeable with C pointers and structures. To take advantage of these F90 capabilities, a second library containing F90 compatible interfaces is located in the file **`ugc_fort90.f90`**. These interfaces are fully compliant with the F90 programming language. The F90 interfaces provide support for dynamic memory allocation and data structure arguments. Only API functions that dynamically allocate memory or pass data structures have an equivalent F90 interface. The names of the F90 interfaces are distinguished with the **`_F90`** extension. To support the C unsigned long data type consistently on 32 and 64 bit architectures, care must be taken in declaring the variable in the Fortran application. An example approach is provided in the file **`ugc_fort90.inc`**.

9.6.4 Sample Plugin Library

The plugin library specification in API V2 was developed as a mechanism for the dynamic loading of (possibly multiple) UGC-compliant meshing libraries. An example plugin library based on the GLib software is included with the distribution, provided in source code form in the **`dist/plugin`** directory. Users can refer to this code for the implementation instructions.

The GLib software package contains the **`gmodule`** library of functions, which provide the functionality to load libraries dynamically at runtime. GLib is available as part of the GIMP Toolkit (GTK+) and may be downloaded freely. Using **`gmodule`** to dynamically load and unload shared libraries is simplified through the use of **`libtool`**, available from the Free Software Foundation's GNU Project.

An outline of the process to incorporate GLib within an application and a API V2-compliant database library is described below. Users can refer to the source code for more detailed information.

Integration into an Application

- **`#include`** the appropriate GLib and **`gmodule`** header files.
- **`#include`** the `ugcplugin.h` header file, which takes care of defining function prototypes for **`gmodule.`**
- Open the library/plugin with **`g_module_open().`**
- Obtain the function pointers with **`g_module_symbol().`**
- Use the function pointers to access the functions from within the application.
- Close the library/plugin using **`g_module_close().`**

Integration into the Database Library

- **`#include`** the **`ugcplugin.h`** header file
- Code the library functions as usual.
- Compile the library using **`libtool`** as shown in the example **`Makefile.`**

To demonstrate the compilation and execution of a dynamically loaded UGC compliant library under the Win32 platform, the sample application and libraries were compiled using Dev-C++, available from Bloodshed Software.

A sample UGC API V2 compliant application (sample_app) and two libraries (module_a and module_b) for Win32 have been provided. These can be referred to for the detail of coding and compiling the GLib**`/gmodule`** functionality into applications and libraries, but the additional steps beyond those for UNIX/Linux are as follows:

For Libraries/Plugins:

- Create a default DLL project in Dev-C++ and add the **`module_(a|b).c,`** **`ugc.h`** and **`ugcplugin.h`** files.
- In **`ugc.h`**, add the following after the **`#define _UGC_H_`** line:
  ```
  #if BUILDING_DLL
  #define DLLIMPORT __declspec(dllexport)
  #else /* Not BUILDING_DLL */
  #define DLLIMPORT __declspec(dllimport)
  #endif /* Not BUILDING_DLL */
  ```
- In the **`ugc.h`** include file, prepend **`DLLIMPORT`** in front of each function declaration so that
  ```
  extern UGC_Status UGC_Session_Inquire_OptionalKeys(char
  ***Keys, UGC_Int *Numkeys);
  ```
  becomes
  ```
  DLLIMPORT extern UGC_Status
  UGC_Session_Inquire_OptionalKeys(char ***Keys, UGC_Int
  *Numkeys);
  ```
- Insert **`DLLIMPORT`** in front of the function names in **`module_(a|b).c`**.

For Applications:

- Create an ordinary Win32 console application in Dev-C++ and add **`sample_app.c, ugcplugin.c, ugc.h,`** **and** **`ugcplugin.h.`**
- Compile and run the application.

10 Conclusions

A general API for unstructured mesh generation has been developed and demonstrated under this contract. This API provides a general specification that standardizes the communication between applications and mesh generation libraries. The design supports a wide range of mesh data types and algorithms. More importantly it provides a general, extensible framework that can be molded to meet future requirements. By itself, the API provides little benefit. The real benefit will be realized when the specification is adopted by developers and integrated into a wide range of mesh generation applications and libraries across the industry. This will allow applications to rapidly integrate new technology utilizing a wide range of mesh generation libraries with little or no modification. The result will speed technology transition and reduce development time.

Widespread adoption of the API specification will not come easily. The API has been designed to minimize the investment required to bring a library or application into compliance; even so, developers and organizations will be reluctant to integrate the API into their software until a critical mass of compliant software is available to justify the cost of implementation. Successful proliferation of the API into a critical mass of software will require encouragement and investment by UGC members and the Air Force.

Several steps can be taken to help advance the API. The specification will need to be publicized so that developers are aware of its existence and the benefits that it offers. This can be accomplished through technical conferences and industry forums. The API documentation must be readily available to the general public and support available to prospective developers. Initial developer support will come from the UGC as a voluntary users group that can answer questions and provide information. The demonstrations provided in this report form the beginning of a repository of examples of API compliant applications and libraries. Additional examples will continue to be developed and added to the repository as developers adopt and implement the specification. These examples will in turn assist future developers.

As more people use the specification and technology continues to evolve, the API specification will undoubtedly need to be modified. A documented process will need to be defined and managed by the UGC for proposing and modifying the public specification. The UGC may also wish to define a process for certifying library and application compliance with the specification.

While sufficient for most applications, the current scope of the specification will restrict its use in some software. Expansion to new meshing operations such as adaptation and mesh partitioning can be accommodated very easily with minor changes to the specification and no changes in the API functions. Some meshing operations such as agglomeration will require the addition of additional mesh elements such as n-sided faces and cells. Expansion to include additional mesh types such as structured, hierarchical and chimera meshes should also be considered in future versions of the API. These types of changes will most likely require the addition of new API functions.

Adoption of the UGC API version 2.0 as an industry standard will radically change the way mesh generation technology is transitioned. With a minor investment to make their software compliant, developers will be able to offer their users a much wider range of capabilities and algorithms than has ever been possible.

11 References

1. International Meshing Roundtable, http://www.imr.sandia.gov/.
2. International Society of Grid Generation, http://www.isg.org/.
3. Object Management Group, Inc., http://www.omg.org/news/about.
4. The CFD General Notation System (CGNS), http://www.cgns.org/.
5. Unstructured Grid Consortium, http://www.pointwise.com/ugc.
6. UGC Standards Document, Version 1.0, 2002, http://www.pointwise.com/ugc/ugcstandv1.pdf.
7. Algorithm Oriented Mesh Database, http://www.scorec.rpi.edu/AOMD.
8. Field Model Library, http://field-model.sourceforge.net/.
9. Grid Algorithms Library, http://www.math.tu-cottbus.de/~berti/gral
10. Tera-Scale Simulation Tools and Technology Center, http://www.tstt-scidac.org/
11. GNU Triangulated Surface Library, http://gts.sourceforge.net/.
12. 3D ACIS Modeler, http://www.spatial.com/.
13. Parasolid, http://www.eds.com/products/plt/parasolid.
14. Open CASCADE, http://www.opencascade.org/.
15. CAPRI: Computational Analysis Programming Tool, http://raphael.mit.edu/capri/docs.html.
16. CAD Services Specification of the Object Management Group, http://mantis.omg.org/mfgcadv1-2rtf.htm.
17. Geometry and Grid Toolkit (GGTK), http://www.eng.uab.edu/me/ETLab/Software/GGTK.
18. Standard Templates Library, http://www.sgi.com/tech/stl.
19. Common Object Model, http://www.microsoft.com/com.
20. Common Component Architecture, http://www.cca-forum.org/software.html.
21. SIDL/Babel, http://www.llnl.gov/CASC/components.
22. PETSc, http://www-fp.mcs.anl.gov/petsc.
23. GLib/GModule documentation, http://www.gtk/org/.
24. Marcum, D.L. and Weatherill, N.P., "Unstructured Grid Generation using Iterative Point Insertion and Local Reconnection," **AIAA Journal**, Vol. 33, No. 9, pp 1619-1625, September 1995.
25. MADCAP.
26. APPT.
27. Doxygen, http://www.doxygen.org/.